KATHOLIEKE UNIVERSITEIT
LEUVEN

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Computer Science

Security of Software on Mobile Devices

Pieter Philippaerts

Dissertation presented in partial
fulfillment of the requirements for
the degree of Doctor
in Engineering

October 2010

Security of Software on Mobile Devices

Pieter Philippaerts

Jury:
Prof. Dr. ir. Paul Van Houtte, president
Prof. Dr. ir. Frank Piessens, promotor
Prof. Dr. ir. Wouter Joosen, promotor
Prof. Dr. ir. Yolande Berbers
Prof. Dr. ir. Bart Preneel
Prof. Dr. Fabio Massacci
Dr. Thomas Walter
Dr. Yves Younan

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

October 2010

D/2010/7515/105
ISBN 978-94-6018-266-2

Abstract

Making sure that a large software system is secure, is a very difficult task. The experience with desktop and server computers teaches us that the insecurity of a system is often proportional to a number of things, including the size of the attack surface (i.e. how easy it is for an attacker to 'talk' to the computer) and how much freedom the user gets to install and execute applications.

Mobile phones are progressing at a rapid and steady pace, making them evermore attractive to attackers. A decade ago, a mobile phone was an electronic brick that was barely capable of supporting telephony functions like calling and messaging contacts. Fast forward to 2010, and they are completely transformed to full-fledged multimedia devices with millions of colors and often touch functionality. Although this evolution is exciting, it also uncovers a number of flaws in the original operating systems and security mechanisms that have not changed much in the last decade.

The main goal of this dissertation is to improve upon the state of the art, in terms of technical security mechanism and measures against attacks on or attacks by software applications running on the mobile phone. There are two different ways to approach this goal. New techniques can be implemented to fix the software that is already out there. The advantage here is that it takes no or only a small effort to update the huge amount of existing applications, but this approach can typically only take you so far. The other option is to rethink everything from scratch. By not having to be backwards compatible, better solutions can be designed and implemented. The disadvantage is that it takes years for the bulk of the software to switch to this new and improved system. However, before suggesting solutions for security problems, we must first show that security is an important topic for mobile phones.

This dissertation presents three main contributions.

1. **Alphanumeric and Turing Complete Shellcode** We prove that in code injection attacks where the attacker is severely limited in the type of data he can use to inject his payload, arbitrary code execution can still be achieved. Hence, intrusion detection systems that are based on the detection of special

patterns could possibly be bypassed by attackers, without limiting the attacker in any way. We show that for the ARM architecture, the subset of instructions that consist of only alphanumeric bytes is Turing Complete. Thus, if an attacker can only use such alphanumeric instructions, he is not limited at all in terms of computational capabilities.

2. **A Countermeasure against Code Injection Attacks** We introduce a new countermeasure to protect applications against code injection attacks. A modified compiler will emit extra security checks during the compilation of a C application. These checks use a masking mechanism to assert that code pointers cannot point to memory areas they should not point to. Even if an attacker can somehow inject shellcode into the memory of an application, the masking process will prevent the attacker from jumping to this injected code.

3. **An Implementation and Evaluation of Security-by-Contract** We have implemented and evaluated a full and efficient implementation of the new Security-by-Contract (SxC) paradigm on the .NET Full and Compact Frameworks. The SxC framework is a deployment framework for managed applications. It tries to statically guarantee that an application will never violate a predetermined system policy, which is designed to protect system resources from abuse. If the framework cannot statically guarantee the compliance of the application with the policy, additional security checks are embedded into the application that enforce compliance.

As the evaluation shows, our tools and frameworks are highly performant and very compatible with existing code. The code injection countermeasure has a performance overhead of only a few percentage points, and a memory overhead of practically 0%. It is also very compatible with existing code bases, as was shown by the SPEC benchmark. The SxC implementation shares the same characteristics. If an application that is being deployed can somehow prove that it will never break the system policy, the SxC framework will not impose any overhead at all. However, for other applications, a runtime monitor is embedded in the application. The performance of this monitor depends almost entirely on the complexity of the policy that is being enforced. Our SxC implementation is very compatible with existing code; not a single application has been reported to not work with the prototype.

Abstract (Dutch)

Het beveiligen van een groot softwaresysteem is een bijzonder moeilijke opdracht. Uit de ervaring met desktop- en servercomputers weten we dat de onveiligheid van een systeem meestal proportioneel is met een aantal dingen, waaronder de grootte van het aanvalsoppervlak (d.w.z. hoe makkelijk het is voor een aanvaller om te 'spreken' met de computer) en hoeveel vrijheid de gebruiker krijgt om applicaties te installeren en uit te voeren.

Mobiele telefoons maken momenteel een bijzonder snelle evolutie mee, waardoor ze vaker als doelwit gezien worden voor aanvallers. In een tijdspanne van slechts enkele jaren zijn mobiele telefoons geëvolueerd van een elektronische baksteen die enkel gebruikt kon worden om te bellen en om berichtjes te sturen, tot een hoogtechnologisch multimediatoestel, waarmee gebruikers kunnen surfen op het internet, films en muziek afspelen, foto's nemen, navigeren door het verkeer, en applicaties installeren. Alhoewel dit een bijzonder interessante evolutie is, brengt ze ook een aantal praktische problemen met zich mee. Omwille van de vele extra functies wordt het ook makkelijker voor aanvallers om een beveiligingslek te vinden dat kan misbruikt worden. Daarnaast zijn de verschillende beveiligingsmaatregelen vaak niet meegeëvolueerd.

Het hoofdopzet van deze thesis is om de huidige stand van de techniek te verbeteren op vlak van beveiligingsmechanismen tegen aanvallen op mobiele toestellen. Dit kan bereikt worden op twee verschillende manieren. Bestaande software kan beveiligd worden door het incorporeren van nieuwe beveiligingstechnieken. Dit heeft als voordeel dat het slechts een kleine moeite vergt om bestaande software te updaten, maar deze aanpak heeft ook zijn beperkingen. Een andere optie is om de software volledige te herschrijven. Dit kan leiden tot veiligere programmacode, maar vereist wel veel meer werk. Echter, vooraleer oplossingen voor diverse beveiligingsproblemen aangedragen worden, moet eerst aangetoond worden dat mobiele toestellen wel degelijk kwetsbaar zijn.

Deze thesis beschrijft drie contributies.

1. **Alfanumerieke en Turingvolledige Shellcode.** We tonen aan dat een aanvaller die een code-injectie-aanval uitvoert, waarbij hij zeer gelimiteerd is in het type van data dat hij kan gebruiken, nog altijd arbitraire code kan uitvoeren. Beveiligingssystemen die speciale gevaarlijke patronen proberen te detecteren kunnen op deze manier omzeild worden, zonder de mogelijkheden van de aanvaller te beperken. We tonen aan dat voor de ARM-architectuur de subset van instructies die bestaan uit alfanumerieke karakters Turingvolledig is. Een aanvaller die enkel gebruik kan maken van dergelijke alfanumerieke instructies is dus niet gelimiteerd in computationele mogelijkheden.

2. **Een Tegenmaatregel voor Code-injectie-aanvallen.** We stellen een nieuwe tegenmaatregel voor die beschermt tegen code-injectie-aanvallen. Een aangepaste compiler zal extra veiligheidscontroles genereren tijden de compilatie van een C-applicatie. Deze controles maken gebruik van een maskeermechanisme dat verzekert dat codewijzers niet naar geheugenadressen kunnen verwijzen waar ze niet naar zouden mogen verwijzen. Indien een aanvaller erin slaagt om nieuwe programmacode in het geheugen van een applicatie te injecteren, zal het maskeermechanisme voorkomen dat de aanvaller de geinjecteerde code kan uitvoeren.

3. **Een Implementatie en Evaluatie van Security-by-Contract.** We hebben een implementatie gemaakt en geëvalueerd van het nieuwe Security-by-Contract (SxC) paradigma op het volledige en het compacte .NET Framework. Het SxC-raamwerk probeert statisch te garanderen dat een applicatie nooit misbruik zal maken van de beschikbare systeembronnen. Als deze garantie niet statisch gegeven kan worden, dan zal het raamwerk extra beveiligingschecks in de code van de applicatie verweven die ervoor zorgen dat misbruik nooit kan voorkomen.

Zoals de evaluatie aantoont, zijn onze beveiligingsmechanismen heel performant en zeer compatibel met bestaande code. De code-injectietegenmaatregel heeft een performantieoverhead van een paar percenten, en heeft een geheugenoverhead van bijna nul percent. Het is zeer compatibel met bestaande code, zoals aangetoond door de SPEC benchmark. De SxC-implementatie deelt dezelfde karakteristieken. Als het systeem statisch kan aantonen dat een applicatie nooit de systeembronnen zal misbruiken, dan zorgt het SxC-systeem voor geen enkele performantie- of geheugenoverhead. Bij applicaties waar dat niet aangetoond kan worden, zullen extra beveiligingschecks worden ingeweven. De overhead van deze extra checks zijn bijna volledig afhankelijk van de complexiteit van de regels die moeten afgedwongen worden. De SxC-implementatie is heel compatibel met bestaande code; er is geen enkele applicatie gevonden die niet bleek te werken met het prototype.

Acknowledgements

First of all, I would like to thank my co-promoter Prof. Dr. ir. Frank Piessens for his guidance, instruction and endless patience. He consistently went above and beyond his commitments as a supervisor and for that I am grateful.

My co-promotor Prof. Dr. ir. Wouter Joosen also deserves a special Thank You. He provided me with the challenge to prove myself, and the opportunity to work at DistriNet.

Many thanks for the interesting and helpful comments on this dissertation go to the members of the jury: Prof. Dr. ir. Paul Van Houtte, Prof. Dr. ir. Yolande Berbers, Prof. Dr. ir. Bart Preneel, Prof. Dr. Fabio Massacci, Dr. Thomas Walter and Dr. Yves Younan.

Finally, I would like to thank all the colleagues who influenced the work in this dissertation either directly or indirectly. At the risk of forgetting someone, these people are: Cédric Boon, Koen Buyens, Prof. Dr. Dave Clarke, Philippe De Ryck, Dr. Bart De Win, Dr. Lieven Desmet, Dominique Devriese, Francesco Gadaleta, Tom Goovaerts, Thomas Heyman, Prof. Dr. Bart Jacobs, Katrien Janssens, Dr. Sven Lachmund, Dr. Bert Lagaisse, Stijn Muylle, Katsiaryna Naliuka, Nick Nikiforakis, Steven Op de beeck, Dr. Davy Preuveneers, Esther Renson, Ghita Saevels, Dr. Riccardo Scandariato, Dr. Jan Smans, Marleen Somers, Raoul Strackx, Steven Van Acker, Dries Vanoverberghe, Ruben Vermeersch, Karen Verresen, Frédéric Vogels, Kim Wuyts, Dr. Yves Younan, and Koen Yskout.

Pieter Philippaerts

October 2010

Contents

Chapter 1

Challenges in Mobile Phone Security

Making sure that a large software system is secure, is a very difficult task. The experience with desktop and server computers teaches us that the insecurity of a system is often proportional to a number of things, including the size of the attack surface (i.e. how easy it is for an attacker to 'talk' to the computer) and how much freedom the user gets to install and execute applications.

Mobile phones are progressing at a rapid and steady pace, making them ever more attractive to attackers. This chapter gives an introduction to the importance of mobile phone security, and lays out the different challenges in this domain. It lists the different goals that will have to be met in order to improve the security of mobile devices, and highlights the contributions that are presented in this dissertation.

1.1 The Importance of Mobile Phone Security

In the timespan of just a few years, mobile phones have gone through an amazing evolution. A decade ago, a mobile phone was an electronic brick that was barely capable of supporting telephony functions like calling and messaging contacts. They were relatively slow, and none of them supported colors. Fast forward to 2010, and they are completely transformed to full-fledged multimedia devices with millions of colors and often touch functionality. They support the playback of music and video, offer support for taking and storing pictures, can connect to the internet, have GPS capabilities, and allow users to install custom applications written by 3rd parties. Although this evolution is exciting, it also uncovers a number of flaws

in the original operating systems and security mechanisms that have not changed much in the last decade.

With hundreds of thousands of applications that can be downloaded and installed on mobile phones, some of them are bound to be malicious in nature and others might be susceptible to attacks. They can leak sensitive information like passwords, emails, photos, or contact information. They can cost the owner of the phone money, if they get access to paid services such as placing a call, sending a short message and surfing the internet. Or they can become part of a botnet, constantly sending SPAM or attacking other computers on the network.

One could argue that infecting a mobile phone is more advantageous for an attacker than infecting a desktop computer. A mobile phone can do almost anything a desktop computer can. Modern phones feature an ARM processor that runs at speeds of up to 1GHz. They often run trimmed down versions of existing desktop operating systems, but this does not hinder the attacker in any way. All the networking and communication capabilities on mobile systems are present or even enhanced, compared to desktop systems.

Mobile phones are also always connected to some kind of network. This can be the cellular network, of course, but in many cases an internet connection (using WiFi, or over the cellular network) or a Bluetooth connection is also available. This increases the attack surface considerably, because all these different types of networks require their own software stacks and applications.

Most mobile phones do not have virus scanners or other malware detection software, and do not have a firewall to make sure that no application can open network connections or starts listening for incoming connections. And even though many phones support some kind of mechanism that allows operators to push or pull software, these mechanisms are typically used to pull spyware from devices, rather than pushing security fixes for the system.

A final problem is that mobile phones easily outnumber desktop and server computers by a factor of 4 to 1. This, combined with the fact that only a small number of different mobile phone operating systems exist, means that if a bug is found in one operating system, hundreds of millions of devices can be exploited.

Because of all these reasons, it is paramount to make sure that all mobile phones come with better security out of the box. This can be achieved in a number of different ways, starting with user and developer education. However, this dissertation will rather focus on the technical frameworks and tools that can be improved upon.

1.2 Challenges

The challenges that are faced on mobile phones are very similar as those on desktop computers. The ramifications are, however, much worse. Security in the field of mobile phones is more difficult and arguably more important than security on desktop computer for a number of reasons.

Security Complacency Security problems on desktop computers are so commonplace, that people assume they are at risk by simply using their computer. They have accepted that things can —and probably will eventually— go wrong. Not so with mobile phones. Because of a relatively good security track record of mobile phones, most people are unaware that mobile devices face the same security challenges that a computer does. This track record will likely worsen as mobile phones get more open and more powerful than ever before, and people will eventually catch up, but by then the damage may have already been done.

Software Vulnerabilities It is almost impossible (and certainly too costly) to write software without bugs. Even the best teams of programmers write buggy software, no matter how much attention to code quality, review of code, or programmer education is given. Some of these programming bugs can be used by an attacker to make the software misbehave. In this case, these bugs are called *vulnerabilities*. Depending on the type of bug, attackers can crash an application or the entire system, influence the control flow of the application (i.e. get access to parts of the program that the attacker should not have gotten access to), or in worst case even allow an attacker to execute arbitrary code.

Users are not security experts The easiest way for an attacker to execute code on a system, is to simply ask the user to run it. Attackers often misdirect the user by influencing the user interface or by somehow encouraging the user to make the wrong decision. Users are no security experts, so they should not be asked to make decisions that impact the security of their devices. In addition, the system should be able to cope with malicious applications that are installed by the user. Under no circumstances should a malicious application get unrestricted access to the resources of a device.

Privacy Violations The privacy of users is particularly at risk when an attacker succeeds in infiltrating a mobile device, more so than on a desktop computer. A mobile phone stores confidential information such as contact information, emails and text messages, notes, photos, ... But if attackers control the system, they can also listen in on phone conversations, track the user through the use of the GPS, intercept and modify text messages, or anything else that the phone is capable of.

Monetary Issues A final challenge that is much more prevalent on mobile phones than on desktop computers, is the fact that an attack can actually cost the user money. If the attacker starts sending text messages or placing phone calls to premium numbers in dodgy countries, or starts sending internet traffic over the cellular network, the mobile operator will charge the user for this.

If we wish to keep the security track record of mobile phones as good as it is today (or even improve upon it), then it is necessary to build additional defenses into mobile operating systems and the associated tools, that offer a better protection than the state of the art. These are the challenges that we address in this dissertation.

1.3 Goals and Approach

The main goal of this dissertation is to improve upon the state of the art, in terms of technical security mechanism and measures against attacks on or attacks by software applications running on the mobile phone. There are two different ways to approach this goal. New techniques can be implemented to fix the software that is already out there. The advantage here is that it takes no or only a small effort to update the huge amount of existing applications, but this approach can typically only take you so far. The other option is to rethink everything from scratch. By not having to be backwards compatible, better solutions can be designed and implemented. The disadvantage is that it takes years for the bulk of the software to switch to this new and improved system.

However, we must first show that security is an important topic for mobile phones. An attacker who takes control over a mobile phone has unlimited access to all the resources of the phone. This is a serious threat, and users often underestimate the danger of this. So, before introducing new technical measures to increase the security of the software stack, we must make the case for implementing new security tools on mobile phones. Our approach is to show that even under harsh conditions for attackers, they might still be able to mount a full and unrestricted attack on the phone.

We want to build tools that can be used by phone manufacturers to secure their applications. The key here is to make sure that it is more difficult for an attacker to exploit a bug in the application. In other words, we want to make sure that bugs do not become vulnerabilities. This can be accomplished in different ways, but our approach is to implement a compiler extension that weaves in additional checks in the binary code during the compilation process. These checks will counter many of the common attack techniques that are used by attackers today.

In addition to minimizing the potential danger of bugs in applications, we also want the phone to be secure in the presence of malicious applications. These applications

are installed and executed by the user, and thus are much harder to detect than other malware. We approach this problem by building a custom framework to deploy applications on the phone. This framework must ensure that applications conform to a predetermined system policy. One possibility is to attach a runtime monitor to the application that is being deployed. By monitoring all the calls to the resources of the phone, an application can be disallowed to abuse the system.

These goals must be met within the technical constraints of a mobile system. Hence, all countermeasures, tools and frameworks must have a very good performance, both in terms of speed overhead, memory overhead and power consumption.

They must also have a relatively low impact on the development process. Since we cannot expect mobile phone manufacturers to rewrite their entire existing code base, our security measures must be applicable without a lot of extra work.

1.4 Context and Technical Challenges

The goals that are listed in Section 1.3 stem from the limitations of existing security architectures and tools. Depending on the tools and frameworks that are used to write an application, different security concerns appear.

All applications can be divided into one of two classes. In the first class, called *unmanaged applications*, the developer is responsible for correctly managing the memory that is allocated to the application. This can be a difficult task, especially in large software systems. One particular disadvantage of unmanaged applications, is that small oversights in the program code might lead to so-called *code injection atttacks*.

For a code injection attack to succeed, the application must contain a bug that gives the attacker the ability to write data outside of the expected memory locations. This can enable the attacker to overwrite interesting memory locations that influence the control flow of the application. If an attacker can then somehow get malicious program code into the memory of the application, he could influence the control flow in such a way that the malicious code is executed. A more detailed description of code injection attacks can be found in Section 2.2.

Some security frameworks for mobile phones require that applications are part of the second class of applications: the so-called *managed applications*. These applications run in a Virtual Machine (VM), which is a sandboxed environment that guarantees a number of things including memory safety. Examples of Virtual Machines are the .NET (Compact) Framework [69], the Java Platform [112], or the Dalvik Virtual Machine [30]. Since attackers will not be able to write outside the bounds of memory structures, they will not be able to influence the application's

control flow and a code injection attack becomes impossible. A more detailed description of virtual machines can be found in Section 2.3.

Attackers must find other ways to attack these applications. One way is to simply mislead users into installing and executing malicious applications. This turns out to be a very successful attack vector, because users are often not very security-aware, and it is extremely difficult for any security architecture to fend off attacks by applications that the user has indicated he wants to run.

1.5 Assumptions and Attack Model

Throughout this dissertation, a number of assumptions are made about the attackers we are trying to protect against. These assumptions can be summed up in a so-called attack model. Section 1.4 introduced the two different classes in which applications can be divided in. Each class has a different attack model.

Unmanaged Applications Writing secure software is difficult, especially when that software is written in an unmanaged programming language. A number of common programming mistakes can lead to successful attacks on the software, where the attacker can force the program to execute arbitrary code of the his choice. The attack model for unmanaged applications models an attacker as someone who tries to gain unauthorized access to a system by trying to exploit a bug in an unmanaged application.

The attack model consists of only one assumption: *the attacker cannot write to code memory.* If the attacker can arbitrarily change the instructions of an application, it is easy to rewrite the application in a way that any security checks are skipped. Non-writable code memory is the standard today, so this assumption certainly does not limit the applicability of our proposed solutions in any way.

The attack model defined here represents a very powerful attacker. In particular, we assume that the attacker can read the contents of the entire memory range of the application, and that data memory can be executed.

Managed Applications Attacking managed applications is more difficult because of the extra security guarantees that are offered by the virtual machine. On a system that requires all applications to be written in a managed language, it is often easier for attackers to trick the user into installing malicious software rather than trying to attack legitimate software. Hence, in the attack model for managed systems, the attacker has succeeded in installing a malicious application and running it like any other normal application.

The only assumption in this attack model is that the virtual machine does not contain bugs. That is, the attacker cannot find a way around the memory and type safety features that are offered by the VM. He can also not exploit bugs in the VM that could lead to native code injection attacks.

The attack model coincides completely with what a normal legitimate application is allowed to do. This immediately implies that exploitable bugs in legitimate applications will also be subject to the same restrictions that are imposed by the security mechanisms we propose.

1.6 Main Contributions

The work in this dissertation focused around application security on mobile devices. Security issues in both native applications and managed applications have been investigated, and solutions have been proposed. The text in this dissertation is structured in a way that reflects the three main contributions.

1.6.1 Alphanumeric and Turing Complete Shellcode

We prove that in code injection attacks where the attacker is severely limited in the type of data he can use to inject his payload, arbitrary code execution can still be achieved. Hence, intrusion detection systems that are based on the detection of special patterns could possibly be bypassed by attackers, without limiting the attacker in any way.

We show that for the ARM architecture, the subset of instructions that consist of only alphanumeric bytes is Turing Complete. Thus, if an attacker can only use such alphanumeric instructions, he is not limited at all in terms of computational capabilities.

This work has been done in the context of a bilateral project between the DistriNet research group and DOCOMO Euro-Labs. It has led to the publication of the following paper:

- *"Filter-resistant code injection on ARM"*, Yves Younan, Pieter Philippaerts, Frank Piessens, Wouter Joosen, Sven Lachmund, and Thomas Walter, Proceedings of the 16th ACM conference on Computer and Communications Security, pages 11-20, November 9-13, 2009

A subset of this work is also published as an article:

- *"Alphanumeric RISC ARM shellcode"*, Yves Younan and Pieter Philippaerts, Phrack Magazine, issue 66, June 11, 2009

1.6.2 A Countermeasure against Code Injection Attacks

We introduce a new countermeasure to protect applications against code injection attacks. A modified compiler will emit extra security checks during the compilation of a C application. These checks use a masking mechanism to assert that code pointers cannot point to memory areas they should not point to. Even if an attacker can somehow inject shellcode into the memory of an application, the masking process will prevent the attacker from jumping to this injected code.

This work has been done in the context of a bilateral project between the DistriNet research group and DOCOMO Euro-Labs. It has led to the following European patent application:

- *"Method and apparatus for preventing modification of a program execution flow"*, Pieter Philippaerts, Yves Younan, Frank Piessens, Sven Lachmund, and Thomas Walter, Application no. 09161239.0-1245, Filing date 27/05/2009

The following paper is currently under submission:

- *"Code Pointer Masking: Hardening Applications against Code Injection Attacks"*, Pieter Philippaerts, Yves Younan, Frank Piessens, Sven Lachmund, and Thomas Walter, submitted to First ACM Conference on Data and Application Security and Privacy (CODASPY) 2011

1.6.3 An Implementation and Evaluation of Security-by-Contract

We have implemented and evaluated a full and efficient implementation of the new Security-by-Contract (SxC) paradigm on the .NET Full and Compact Frameworks. The SxC framework is a deployment framework for managed applications. It tries to statically guarantee that an application will never violate a predetermined system policy, which is designed to protect system resources from abuse. If the framework cannot statically guarantee the compliance of the application with the policy, additional security checks are embedded into the application that enforce compliance.

This work has been done in the context of the European FP6 project *'Security of Software and Services for Mobile Systems (S3MS)'*. It has led to the publication of the following papers:

- *"A flexible security architecture to support third-party applications on mobile devices"*, Lieven Desmet, Wouter Joosen, Fabio Massacci, Katsiaryna Naliuka, Pieter Philippaerts, Frank Piessens, and Dries Vanoverberghe, in Proceedings of the 2007 ACM workshop on Computer security architecture, pages 19-28, November 2, 2007
- *"Security-by-contract on the .NET platform"*, Lieven Desmet, Wouter Joosen, Fabio Massacci, Pieter Philippaerts, Frank Piessens, Ida Siahaan, and Dries Vanoverberghe, in Information security technical report, volume 13, issue 1, pages 25-32, May 15, 2008
- *"Security middleware for mobile applications"*, Bart De Win, Tom Goovaerts, Wouter Joosen, Pieter Philippaerts, Frank Piessens, and Yves Younan, in Middleware for network eccentric and mobile applications, pages 265-284, 2009
- *"A security architecture for Web 2.0 applications"*, Lieven Desmet, Wouter Joosen, Fabio Massacci, Katsiaryna Naliuka, Pieter Philippaerts, Frank Piessens, Ida Siahaan, and Dries Vanoverberghe, in Towards the Future Internet - A European Research Perspective, 2009
- *"The S3MS.NET run time monitor: Tool demonstration"*, Lieven Desmet, Wouter Joosen, Fabio Massacci, Katsiaryna Naliuka, Pieter Philippaerts, Frank Piessens, and Dries Vanoverberghe, in Electronic Notes in Theoretical Computer Science, volume 253, issue 5, pages 153-159, December 1, 2009

1.6.4 Other Contributions

During the author's PhD studies other research work has been done that is not reported on in this dissertation. The author's input in this work ranges from significant contributions to mere guidance and supervision.

Cryptographic Extensibility in .NET The extensibility of the .NET cryptographic API has been investigated in [85] and [18]. A number of design issues have been uncovered, and backwards compatible suggestions have been offered.

Breaking the Memory Secrecy Assumption A number of countermeasures against code injection attacks are built on the assumption that application memory is secret and can never be read by an attacker. This myth is debunked in [111].

An Efficient Pointer Arithmetic Checker for C Programs A new countermeasure is proposed in [125] to protect against buffer overflows in C programs. It works by

instrumenting C-code with runtime checks that ensure no out-of-bounds calculations occur during pointer arithmetic.

Hardware-optimized Buffer Overflow Protection A port of the existing Multi-Stack countermeasure [126] from the x86 architecture to the ARM architecture is described in [110]. By taking advantage of some of the features of the ARM processor, the memory that is required to run the countermeasure can be reduced severely.

1.7 Structure of this Dissertation

The structure of the rest of this dissertation is as follows. Chapter 2 gives a short introduction to the ARM processor, code injection attacks and security architectures for mobile devices.

Chapter 3 proves that in code injection attacks where the attacker is severely limited in the type of data he can use to inject his payload, arbitrary code execution can still be achieved.

In Chapter 4, we introduce a new countermeasure to protect applications against code injection attacks. A modified compiler introduces a masking mechanism to assert that code pointers cannot point to memory they should not point to.

Our implementation of the Security-by-Contract paradigm on the .NET Framework is described in Chapter 5. This prototype monitors the access of applications to valuable resources, and optionally blocks access to these resources if the monitor deems this necessary.

Finally, Chapter 6 concludes this dissertation.

Chapter 2

Background

To understand the contributions in this dissertation, some background knowledge is assumed. This chapter gives an introduction to a selected number of more advanced background topics that are specific to either mobile devices or application security.

Each section of this chapter introduces one particular topic that is important to understand the rest of this dissertation. The different sections are independent from each other, so the reader may pick and choose the different sections he wishes to read, depending on his background.

Section 2.1 introduces the ARM architecture, which is the dominant processor architecture for mobile devices. It gives a general overview of the architecture, and zooms in on some specific features that are important to understand the contributions in Chapter 3 and Chapter 4. Detailed information about the ARM architecture can be found in the ARM Architecture Reference Manual [51].

Section 2.2 explains the basics of code injection attacks, and summarizes a number of countermeasures that have been suggested to try to stop these attacks. Code injection attacks will be the main theme of Chapter 4. A much more complete overview of memory management errors and countermeasures can be found in [122].

Finally, Section 2.3 highlights the most common mobile security architectures. This serves as general background information for Chapter 5, where we introduce a new security architecture for mobile devices. Resources with detailed information about the existing architectures are cited in the different subsections of Section 2.3.

Register	Purpose
`r0` to `r3`	Temporary registers
`r4` to `r10`	Permanent registers
`r11` (alias `fp`)	Frame pointer
`r12` (alias `ip`)	Intra-procedure call scratch register
`r13` (alias `sp`)	Stack pointer
`r14` (alias `lr`)	Link register
`r15` (alias `pc`)	Program counter

Table 2.1: The different general purpose ARM registers, and their intended purpose.

2.1 The ARM Architecture

The ARM architecture [102] is the dominating processor architecture for cell phones and other embedded devices. It is a 32-bit RISC architecture developed by ARM Ltd. and licensed to a number of processor manufacturers. Due to its low power consumption and architectural simplicity, it is particularly suitable for resource constrained and embedded devices. It absolutely dominates the market: 98% of mobile devices feature at least one ARM processor [25].

ARM processors have traditionally been based on the Von Neumann architecture, where data and code were stored in the memory regions. Starting from the ARM9 cores, the design has started moving to a Harvard architecture with separated caches for code and data, which allows for higher clock frequencies.

2.1.1 Registers

The ARM processor features sixteen general purpose registers, numbered `r0` to `r15`. Apart from the program counter register, `r15` or its alias `pc`, all registers can be used for any purpose. There are, however, conventional roles assigned to some particular registers. Table 2.1 gives an overview of the registers, their purpose, and their optional alias. In addition to these general purpose registers, ARM processors also contain *the Current Program Status Register (CPSR)*. This register stores different types of flags and condition values. This register cannot be addressed directly.

2.1.2 Instructions

The ARM architecture has an atypically extensive and diverse set of instructions for a RISC architecture. Instructions are pipelined into a multi-stage decoding and

execution phase. Older ARM processors used a three-stage pipeline, whereas newer versions use a pipeline of up to nine stages.

ARM processors can be enhanced with various co-processors. Each co-processor offers its own set of instructions, and can be plugged seamlessly into the overall architecture. If the main processor encounters an instruction that it does not recognize, it sends the instruction to a co-processor bus to be executed. If none of the co-processors recognize this instruction, the main processor raises an 'invalid instruction' exception.

2.1.3 Function Calls

Due to the large number of registers, the ARM application binary interface stipulates that the first four parameters of a function should be passed via registers `r0` to `r3`. If there are more than four parameters, the subsequent parameters will be pushed on the stack. Likewise, the return address of a function is not always pushed on the stack. The `BL` instruction, which calculates the return address and jumps to a specified subroutine, will store the return address in register `lr`. It is then up to the implementation of that subroutine to store the return address on the stack or not.

2.1.4 Addressing Modes

ARM instructions share common ways to calculate memory addresses or values to be used as operands for instructions. These calculations of memory addresses are called *addressing modes*. A number of different addressing modes exist, some of which will be explained in this section.

The ARM architecture is a 32-bit architecture, hence it is imperative that the operands of instructions must be able to span the entire 32-bit addressing range. However, since ARM instructions are 32 bits and a number of these bits are used to encode the instruction OP code, operands and parameters, operands that represent immediate values will never be able to store a full 32-bit value. To overcome this problem, some addressing modes support different types of shifts and rotations. These operations make it possible to quickly generate large numbers (via bit shifting), without having to specify them as immediate values.

The following paragraphs will describe a number of addressing modes that are used on ARM. These addressing modes are selected because they will be used extensively in the rest of this dissertation.

Addressing modes for data processing The first type of addressing mode is the mode that is used for the data processing instructions. This includes the instructions that perform arithmetic operations, the instructions that copy values into registers, and the instructions that copy values between registers.

In the general case, a data processing instruction looks like this:

```
<instruction> <Rd>, <Rn>, <shifter_operand>
```

In this example, *Rd* is a placeholder for the destination register, and *Rn* represents the base register.

The addressing mode is denoted in the above listing as the *shifter_operand*. It is twelve bits large and can be one of eleven subcategories. These subcategories perform all kinds of different operations on the operand, such as logical and arithmetic bit shifts, bit rotations, or no additional computation at all. Some examples are given below:

```
mov  r1, #1
add  r5, r6, r1, LSL #2
sub  r3, r5, #1
mov  r0, r3, ROR r1
```

The first `MOV` instruction simply copies the value one into register `r1`. The form of the `MOV` instruction is atypical for data processing instructions, because it does not use the base register *Rn*.

The `ADD` instruction uses an addressing mode that shifts the value in `r1` left by two places. This result is added to the value stored in base register `r6`, and the result is written to register `r5`.

The `SUB` instruction uses the same addressing mode as the first `MOV` instruction, but also uses the base register *Rn*. In this case, the value one is subtracted from the value in base register `r5`, and the result is stored in `r3`.

Finally, a second `MOV` operation rotates the value in `r3` right by a number of places as determined by the value in `r1`. The result is stored in `r0`.

Addressing modes for load/store The second type of addressing mode is used for instructions that load data from memory and store data to memory. The general syntax of these instructions is:

```
<LDR instr> <Rd>, addr_mode
<STR instr> <Rd>, addr_mode
```

The *addr_mode* operand is the memory address where the data resides, and can be calculated with one of nine addressing mode variants. Addresses can come from

immediate values and registers (potentially scaled by shifting the contents), and can be post- or pre-incremented.

Addressing modes for load/store multiple The third type of addressing mode is used with the instructions that perform multiple loads and stores at once. The `LDM` and `STM` instructions take a list of registers, and will either load data into the registers in this list, or store data from these registers in memory. The general syntax for multiple loads and stores looks like this:

```
<LDM instr><addr_mode> <Rn>{!}, <registers>
<STM instr><addr_mode> <Rn>{!}, <registers>
```

The *addr_mode* operand can be one of the following four possibilities: *increment after (IA)*, *increment before (IB)*, *decrement after (DA)*, or *decrement before (DB)*. In all cases, *Rn* is used as the base register to start computing memory addresses where the selected registers will be stored or loaded. The different addressing modes specify different schemes of computing these addresses.

When the optional exclamation mark after the base register is present, the processor will update the value in *Rn* to contain the newly computed memory address.

2.1.5 Conditional Execution

Almost every instruction on an ARM processor can be executed conditionally. The four most-significant bits of these instructions encode a condition code that specifies which condition should be met before executing the instruction. Prior to actually executing an instruction, the processor will first check the `CPSR` register to ensure that its contents corresponds to the status encoded in the condition bits of the instruction. If the condition code does not match, the instruction is discarded.

The `CPSR` state can be updated by calling the `CMP` instruction, much like on the Intel x86 architecture. This instruction compares a value from a register to a value calculated in a *shifter_operand* and updates the `CPSR` bits accordingly. In addition to this, every other instruction that uses the addressing mode for data processing can also optionally update the `CPSR` register. In this case, the result of the instruction is compared to the value 0.

When writing ARM assembly, the conditional execution of an instruction is represented by adding a suffix to the name of the instruction that denotes in which circumstances it will be executed. Without this suffix, the instruction will always be executed. If the instruction supports updating the `CPSR` register, the additional suffix *'S'* indicates that the instruction should update the `CPSR` register.

The main advantage of conditional execution is the support for more compact program code. As a short example, consider the following C fragment:

```
if (err != 0)
    printf("Errorcode = %i\n", err);
else
    printf("OK!\n");
```

By default, GCC compiles the above code to:

```
  cmp   r1, #0
  beq   .L4
  ldr   r0, <string_1_address>
  bl    printf
  b     .L8
.L4:
  ldr   r0, <string_2_address>
  bl    printf
.L8:
```

The value in `r1` contains the value of the *err* variable, and is compared to the value 0. If the contents of `r1` is zero, the code branches to the label *.L4*, where the string *'OK!'* is printed out. If the value in `r1` is not zero, the `BEQ` instruction is not executed, and the code continues to print out the *ErrorCode* string. Finally, it branches to label *.L8*.

With conditional execution, the above code could be rewritten as:

```
  cmp   r1, #0
  ldrne r0, <string_1_address>
  ldreq r0, <string_2_address>
  bl    printf
```

The *'NE'* suffix means that the instruction will only be executed if the contents of, in this case, `r1` is not equal to zero. Similarly, the *'EQ'* suffix means that the instructions will be executed if the contents of `r1` is equal to zero.

2.1.6 Thumb Instructions

In order to further increase code density, most ARM processors support a second instruction set called the Thumb instruction set. These Thumb instructions are 16 bits in size, compared to the 32 bits of ordinary ARM instructions. Prior to ARMv6, only the T variants of the ARM processor supported this mode (e.g. ARM4T). However, as of ARMv6, Thumb support is mandatory.

Instructions executed in 32-bit mode are called ARM instructions, whereas instructions executed in 16-bit mode are called Thumb instructions. Unlike ARM instructions, Thumb instructions do not support conditional execution.

2.2 Code Injection Attacks

Code injection attacks occur when an attacker can successfully divert the processor's control flow to a memory location whose contents is controlled by an attacker. The only way an attacker can influence the control flow of the processor is by overwriting locations in memory that store so-called *code pointers*. A code pointer is a variable that contains the memory address of a function or some other location in the application code where the processor will at some point jump to. Well-known code pointers are the return address and function pointers.

There is a wide variety of techniques to achieve this, ranging from the classic stack-based buffer overflow, to virtual function pointer overwrites, indirect pointer overwrites, and so forth. An example of such an attack on a mobile phone is Moore's attack [72] against the Apple iPhone. This attack exploits LibTIFF vulnerabilities [81, 82], and it could be triggered from both the phone's mail client and its web browser, making it remotely exploitable. A similar vulnerability was found in the way GIF files were handled by the Android web browser [83].

In this section, we briefly describe the most basic type of code injection attack, which occurs by writing outside the bounds of a buffer on the stack and overwriting the return address. This type of attack can no longer be exploited in most cases, due to the deployment of various countermeasures. However, it is very easy to explain, and thus serves as a perfect illustration of the basics of a code injection attack. We then discuss the widely deployed countermeasures, and also explain more advanced attack techniques that can be used to get around these countermeasures.

2.2.1 Stack-based Buffer Overflows

When an array is declared in C, space is reserved for it and the array is manipulated by means of a pointer to the first byte. No information about the array size is available at runtime, and most C-compilers will generate code that will allow a program to copy data beyond the end of an array, overwriting adjacent memory space. If interesting information is stored somewhere in the adjacent memory space, it could be possible for an attacker to overwrite it. On the stack this is usually the case: it stores the addresses to resume execution after a function call has completed its execution, i.e., the return address.

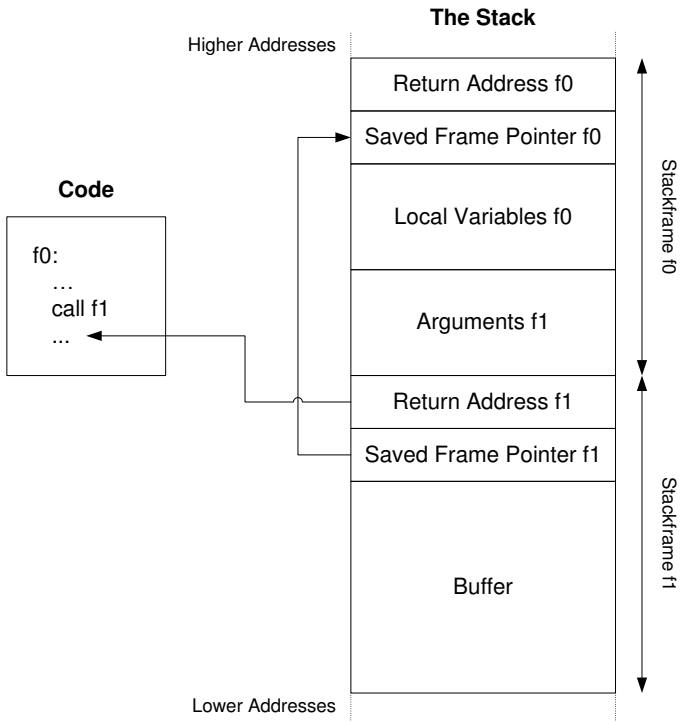


Figure 2.1: A typical stack layout with two functions *f0* and *f1*

For example, on the x86 and ARM architectures the stack grows down (i.e., newer function calls have their variables stored at lower addresses than older ones). The stack is divided into stackframes. Each stackframe contains information about the current function: arguments of the called function, registers whose values must be stored across function calls, local variables and the return address. This memory layout is shown in Figure 2.1. An array allocated on the stack will usually be located in the section of local variables of a stackframe. If a program copies data past the end of this array, it will overwrite anything else stored before it and thus will overwrite other data stored on the stack, like the return address [4].

If an attacker can somehow get binary code in the application's memory space, then he can use the above technique to overwrite a return address and divert the control flow to his binary code that is stored somewhere in the application's memory. This binary code is called *shellcode*, and is typically a very short code fragment that seeks to allow the attacker to execute arbitrary instructions with the same privilege level as the application. A common way of getting this shellcode in the memory space of the application is by giving it as input to the application. This input is

then typically copied to the stack or the heap, where the attacker can then divert the control flow to.

2.2.2 Countermeasures and Advanced Attacks

The stack-based buffer overflow attack described above is the oldest and best-known code injection attack. However, more advanced attack techniques follow a similar pattern in the sense that at some point a code pointer gets overwritten.

Stack Canaries Stack canaries were introduced specifically to counter the threat of stack-based buffer overflows [27, 29]. The countermeasure works by introducing a secret random value on the stack, right before the return address. If an attacker uses a stack-based buffer overflow to overwrite the return address, he will also have to overwrite the canary that is placed between the buffer and the return address. When a function exits, it checks whether the canary has been corrupted. If so, the application is killed before the modified return address is used.

The initial implementations of stack canaries were foiled by using indirect pointer overwrite attacks. The attack worked if an attacker was able to overwrite a pointer and an integer value that were located on the stack between the buffer that has been overflown and the canary. If the application code later uses that pointer to store the value of the integer somewhere in memory, the return address can still be overwritten without invalidating the canary. The attacker can overwrite the pointer to point to the memory location that contains the return address, and he can overwrite the integer with the address of his shellcode. When the application later uses the pointer and writes the value of the integer at the dereferenced location, it effectively overwrites the return address with the address of the attacker's shellcode. ProPolice [42] is the most popular variation of the stack canaries countermeasure. It reorders the local variables of a function on the stack, in order to provide protection against indirect pointer overwrite attacks.

However, even ProPolice is still vulnerable to information leakage such as buffer-overreads [111] and format string vulnerabilities [62]. Since ProPolice only protects stack frames, it will not stop attacks that do not target the stack (for example, heap-based buffer overflows).

ProPolice will also not emit the canary for every function. Only functions that manipulate buffers (and hence are vulnerable to a buffer overflow attack) are protected this way. However, sometimes the heuristics in the compilers are wrong and do not protect functions that *are* vulnerable to attack[1]. In this case, it is trivial for the attacker to overwrite the return address [105].

[1] CVE-2007-0038

Even in the absence of circumstances that allow an attacker to circumvent the protection of ProPolice, only the stack is protected against buffer overflows. However, buffer overflows in other memory regions like the heap [7] or the data segment [91] are also possible. Attackers have also been able to exploit format string vulnerabilities [97], dangling pointer references [33] and integer errors [17] to achieve similar results.

Address Space Layout Randomization The aim of Address Space Layout Randomization (ASLR) [13] is not to stop buffer overflows, but rather to make it much harder for attackers to exploit them. By randomizing the base address of important structures such as the stack, heap, and libraries, attackers will have difficulties finding their injected shellcode in memory. Even if they succeed in overwriting a code pointer, they will not know where to point it to.

ASLR is an excellent countermeasure to have, especially because it raises the security bar with no performance cost. However, there are different ways to get around the protection it provides. Like all countermeasures that rely on random or secret values, ASLR is susceptible to information leakage. In particular, buffer-overreads [111] and format string vulnerabilities [62] are a problem for ASLR.

On 32-bit architectures, the amount of randomization is not prohibitively large [99]. This enables attackers to successfully perform a code injection attack by simply guessing addresses. Eventually, the attack will succeed. New attacks also use a technique called heap-spraying [46, 87, 101]. Attackers pollute the heap by filling it with numerous copies of their shellcode, and then jump to somewhere on the heap. Because most of the memory is filled with their shellcode, there is a good chance that the jump will land on an address that is part of their shellcode.

Non-executable memory Modern processors often support marking memory pages as non-executable. Even if the attacker can inject shellcode into the application and jump to it, the processor would refuse to execute it. There is no performance overhead when using this countermeasure, and it raises the security bar quite a bit.

There are some problems with this countermeasure though. Some processors still do not have this feature, and even if it is present in hardware, operating systems do not always turn it on by default. Linux supports non-executable memory, but many distributions do not use it, or only use it for some memory regions. A reason for not using it, is that it breaks applications that expect the stack or heap to be executable.

But even applications that use non-executable memory are vulnerable to attack. Instead of injecting code directly, attackers can inject a specially crafted fake stack. If the application starts unwinding the stack, it will unwind the fake stack

instead of the original calling stack. This allows an attacker to direct the processor to library functions and choose which parameters are passed to these functions, giving him effectively unlimited control over the processor. This type of attack is referred to as a *return-into-libc* attack [119]. A related attack is called *return-oriented programming* [98], where a similar effect is achieved by filling the stack with return addresses to specifically chosen locations in code memory that execute some instructions and then perform a return. Other attacks exist that bypass non-executable memory by first marking the memory where they injected their code as executable, and then jumping to it [6, 100], or by disabling non-executable memory altogether [105].

Other Countermeasures Many other countermeasures have been developed to counter the threat of buffer overflows and code injection attacks [40, 123], but have not found their way into the mainstream. These countermeasures typically offer a performance/security trade-off that is too small to justify their use. Either they offer a good protection against attacks but perform badly, or they perform very well but offer too little extra security benefits.

Some countermeasures aim to prevent the vulnerability from becoming exploitable by verifying that an exploitation attempt has occurred: via bounds checking [53, 94, 125]. Others will make it harder for an attacker to execute injected code by randomizing the base address of memory regions [15], encrypting pointers [28], code [10, 54] or even all objects [14] while in memory and decrypting them before use. While yet other types of countermeasures will try and ensure that the program adheres to some predetermined policy [1, 56, 86].

Attackers have found ways of bypassing many of these countermeasures. These bypasses range from overwriting control flow information not protected by the countermeasure [21, 90], to guessing or leaking the secret associated with countermeasures [99, 107, 111], to executing existing code rather than injecting code [20, 98, 103, 119], to performing intricate attacks that make use of properties of higher level languages (like JavaScript in the webbrowser) to create an environment suitable to exploit a low-level vulnerability [106].

2.3 Mobile Security Architectures

The security architecture of a mobile phone is the key component that will try to mitigate, if not avoid, any harm that a malicious application can inflict. However, due to the constraints in processing power, they typically cannot be as elaborate as security architectures on modern desktop computers. In addition, the user model is also quite different, because mobile phones are oriented towards a single user,

whereas desktop PCs support multiple user accounts. Hence, new architectures must be designed to cope with these limitations and characteristics.

Even the older phone operating systems supported some kind of application security mechanisms. This mechanism, albeit relatively primitive, has developed into the de facto standard of the industry. Only recently have newer approaches been introduced, Android's being one of the more interesting ones. In this section, we discuss both the traditional mechanism and the newer mechanisms introduced with more recent phones.

2.3.1 Application Signing

The traditional way of doing application security on mobile phones, is called *application signing*. It is the dominant security architecture, and is present in Symbian devices and Windows CE devices. Because of the domination of Symbian-based phones, this technique is often referred to as *Symbian Signing* [64]. For the remaining of this section, we will look at how it is implemented on Windows CE, but the implementations on other systems are very similar.

The security system of Windows CE [71] can run in different modes, depending on the needs of the user. In the more secure 'two-tier' mode, applications can be run in a trusted or partially trusted context. In a trusted context, an application runs unrestricted, whereas in a partially trusted context it is prohibited from accessing a predefined set of sensitive services, thus limiting the amount of damage it can do. However, partially trusted applications are still allowed to access a number of APIs that can cause damage to the system or to the user. For instance, they can make phone calls or send short messages without limitation.

In Windows CE, applications can be signed with a digital signature, or unsigned. When an application is started, the operating system checks the digital signature on the executable. If the signature traces back to a certificate in the trusted certificate store, the application is executed in a trusted context. Likewise, if the signature traces back to a certificate in the untrusted certificate store, the application is executed in a partially trusted context. If the signer is unknown, if the signature is invalid, or if no signature is present, the operating system will ask the user whether the application should be executed. When the user indicates that the system should run the application, it is executed in a partially trusted context.

This signature-based system does not work well for modern phones where users can install any application written by anyone. The first problem is that the decision of allowing an application to run or not, is too difficult for a user to make. She would like to run an application as long as the application does not misbehave or does not violate some kind of policy. But she is not in a position to know what the

downloaded application exactly does, so she cannot make an educated decision to allow or disallow the execution of a program.

A second problem is that certifying an application by a trusted third party is rather expensive. Many of the mobile application developers are small companies that do not have the resources to certify their applications.

A third, and perhaps most damning, problem is that these digital signatures do not have a precise meaning in the context of security. They confer some degree of trust about the origin of the software, but they say nothing about how trustworthy the application is. Cases are already known where malware was signed by a commercial trusted third party [43, 88]. This malware would have no problems passing through the Windows CE security architecture, without a user noticing anything.

This security architecture does a decent job when it is rare that new applications are installed on a phone, and when there are only a very limited number of application developers. But it completely breaks down in the new environment where there are tens of thousands of application developers. Also, the form of sandboxing that it offers is by far too coarse grained to be useful in modern scenarios.

2.3.2 Sandboxing with Virtual Machines

Code injection attacks through memory management vulnerabilities have been around for decades, and security architectures such as application signing do not stop these attacks at all. A number of solutions and countermeasures have been proposed, as discussed in Section 2.2. However, these proposals are typically too slow or only offer partial protection. After almost 25 years of research, it turns out that it seems impossible to find a good solution that remains fully compatible with existing code.

In order to completely solve the problem, a redesign of the C programming language is necessary to make it safe. These safe languages, also called *managed* languages, try to increase software reliability and security by offering extra guarantees about the state of the memory and about how the application can interact with it. In particular, type safety is checked at compile time and runtime, and extra checks are emitted in the application code to make sure out of bound memory operations are not possible. In addition, the use of pointers is severely constrained. These changes incur a performance hit and are not backwards compatible, but they offer (provable) full protection against memory management errors.

Managed languages require a software infrastructure in order to work, called a *virtual machine (VM)*. The virtual machine offers a sandboxed environment for applications to run in. Because of the memory safety guarantees, applications will not be able to break out of their sandbox. In addition, a virtual machine also often offers extra services such as garbage collection.

The two most popular virtual machines on desktop computers are the Java Framework and the .NET Framework. Both platforms are also available for mobile phones (called respectively *Java Micro Edition*, and *the .NET Compact Framework*), albeit in a stripped down version. There is a trend in the mobile phone industry to force third party developers to use one of these frameworks. Starting from Windows Phone 7 Series, Microsoft requires all third party applications to be written in one of the .NET languages that target the .NET Compact Framework. Similarly, all applications on the Android operating system must be written to run on the Dalvik virtual machine.

Executing applications in a virtual machine solves all the problems related to memory management vulnerabilities, however it does not solve all security problems. Even though an application will not be able to do whatever it likes, there are still a number of things that they can do that can potentially cause damage to the owner of the mobile phone in some way. Examples of this behavior might be sending short messages or calling toll numbers without the user's knowledge, listening in on conversations and sending them to a third party, stealing or destroying documents stored on the phone, ... Although the full version of the .NET Framework has a mechanism, called Code Access Security (CAS, [44]), to limit the capabilities of specific applications, this security measure is not implemented on the .NET Compact Framework for performance reasons. However, applications *can* use the signing mechanismn presented in Section 2.3.1, which is effectively a very coarse grained alternative.

2.3.3 The Mobile Information Device Profile

In contrast to the .NET Compact Framework, the Java Micro Edition *has* a mechanism to limit the capabilities of specific applications. Java's *Mobile Information Device Profile (MIDP)* [113] for Java ME specifies a security architecture that relates to Symbian signing, but is more fine grained.

MIDlets — applications that adhere to the MIDP specification — can be assigned a set of permissions. These permissions are defined by the class library, and can be extended as needed. The set of permissions that is assigned to a particular MIDlet is called a *protection domain*. At runtime, the different security-relevant functions of the Java ME Framework check whether the MIDlet has the appropriate permissions in its protection domain before executing. If the permissions are found, the function call is executed and the result is returned. If the permissions are not found, the user is asked to make a decision. He can choose to allow the function call once, all the time, or never. If the user chooses to not allow the call, a SecurityException is thrown.

The MIDP framework supports the creation of multiple protection domains, each with their own predefined set of permissions. When a MIDlet is started, it is

automatically assigned to one of these protection domains, depending on its digital signature. If no digital signature is present, the MIDlet is assigned the 'Untrusted' protection domain. In this case, the user is consulted for all permission assignments.

There are a number of differences between Symbian signing and the MIDP approach. With Symbian signing, the permissions are assigned at program startup. Any and all user interaction takes place at that point. The permission sets are hard-coded in the operating system, and are not extensible. The MIDP framework on the other hand will ask for user input at the point that the permission is actually needed. Permission sets are not hard-coded and can be extended by class library developers.

2.3.4 Android's Permissions-based Scheme

Android [47] is the mobile phone platform that has been introduced late 2008 by a consortium led by Google. It aims to replace all the different software stacks that each mobile operator has, and substitute it with a single platform. Android has an open philosophy, which means that it is open source and that they encourage users to write applications for it and publish them.

The Android software stack runs on top of a Linux kernel, specially tuned for mobile devices. The software stack consists of a set of libraries and a Java-like virtual machine called *Dalvik*. All applications are run inside this virtual machine, and can access the different libraries through a managed Java interface.

In addition to the major memory safety advantages, the Virtual Machine also offers a permission-based system for critical APIs [38]. Like in the full Java framework, the permission-based system checks all incoming requests into the library and can optionally reject them if the calling application does not have the required privileges.

As a consequence of the limited capabilities of mobile phones, Android's permission system does not implement the stack inspection mechanism that is present in the full Java framework. It does, however, feature a similar but simplified system. Applications run without permissions by default. They can request new permissions by specifying them in the *AndroidManifest.xml* file that is associated with the application. During the installation of the application, the system either grants the requested permissions or not, depending on the signature of the installation package or user interaction. Permissions are granted *only* at installation time; no changes to the permission set are made when the application is started. When an application is running and tries to access a privileged method, the virtual machine checks whether the application has the necessary permission that is required to perform the action. If it does not have the permission, an exception is thrown.

All applications on Android must be signed with a digital signature. However, in contrast with the approach described in Section 2.3.1, the signature on Android does not imply that an application is safe to install or not. Instead, it is used solely to determine which applications are developed by the same person or company. Hence, the signature certificates are not required to be published by commercial certificates authorities, as self-signed certificates will fit the required purpose perfectly.

Each Android application runs under a unique Linux user ID. This enforces that different programs cannot access each other's data. If two applications need to share the same data, they can request to share a common user ID. Only programs that are signed by the same developer are allowed to share a user ID.

The Android security model is much more granular than the application signing approach. Even if the user allows an application to be installed on the system, the application does not necessarily get access to all the functionality of the device. At most, it gets access to the functionality it requests. And even this can be limited further by the user.

2.3.5 Advanced Security Architectures

A number of other security architectures and tools have been proposed in the literature to guarantee the security of mobile devices. F-Secure is one of the anti-virus companies that sells a mobile version of their software for Windows Mobile, Symbian and Android. Bose et al. [19] developed a mechanism based on support vector machines, where application behavior is analyzed and compared to the behavior of mobile viruses. When the system detects malicious behavior, the application is killed. Cheng et al. [22] and Schmidt et al. [96] propose systems where the communication behavior is monitored and sent to a central server. This server then analyzes the data and compares it to data of other devices. In this way, an epidemic can easily be detected.

Muthukumaran et al. [74] and Zhang et al. [127] build on the concepts that are present in SELinux to build a secure mobile phone software architecture. They demonstrate that it is possible to write security policies and protect critical applications from untrusted code. Becher et al. [12] implemented a security system on Windows CE, based on kernel-level function call interception. It can be used to enforce more expressive policies than those that are allowed by the default Windows CE security architecture.

Enck et al. [37] improve the existing Android security architecture by detecting requests for dangerous combinations of permissions. When an application is installed, their Kirin system matches the requested permissions with a set of permission combinations that are considered to be dangerous. Ongtang et al. [80] further improve the Android security architecture with runtime checks.

Chapter 3

Filter-resistant Shellcode

With the rapid spread of mobile devices, the ARM processor has become the most widespread 32-bit CPU core in the world. ARM processors offer a great trade-off between power consumption and processing power, which makes them an excellent candidate for mobile and embedded devices. About 98% of mobile phones and personal digital assistants feature at least one ARM processor. The ARM architecture is also making inroads into more high-end devices, such as tablet PCs, netbooks, and in the near future perhaps even servers [109].

Only recently, however, have these devices become powerful enough to let users connect over the internet to various services, and to share information as we are used to on desktop PCs. Unfortunately, this introduces a number of security risks: mobile devices are more and more subject to external attacks that aim to control the behavior of the device.

A very important class of such attacks is code injection attacks (see Section 2.2). Attackers abuse a bug in an application to divert the processor from the normal control flow of the application. Typically, the processor is diverted to shellcode, which is often placed in the memory of the application by sending it as input to the application.

A common hurdle for exploit writers, is that the shellcode has to pass one or more filtering methods before being stored into memory. The shellcode enters the system as data, and various validations and transformations can be applied to this data. An example is an input validation filter that matches the input with a given regular expression, and blocks any input that does not match. A popular regular expression for example is [a-zA-Z0-9] (possibly extended by "space"). Other examples are text encoding filters that encode input into specific character encodings (e.g. UTF-8), and filters that make sure that the input is of a specific type (e.g. valid HTML).

Clearly, for a code injection attack to succeed, the shellcode must survive all these validations and transformations. The key contribution of this chapter is that it shows that it is possible to write powerful shellcode that passes such filters. More specifically, we show that the subset of ARM machine code programs that (when interpreted as data) consist only of alphanumerical characters (i.e. letters and digits) is a Turing complete subset. This is a non-trivial result, as the ARM is a RISC architecture with fixed width instructions of 32 bits, and hence the number of instructions that consist only of alphanumeric characters is very limited.

The research presented in this chapter has been conducted in the context of a bilateral project between the DistriNet research group and DOCOMO Euro-Labs. The contents of this chapter builds on the general introduction to the ARM architecture presented in Section 2.1.

3.1 Alphanumeric Instructions

In most cases, alphanumeric bytes are likely to get through conversions and filters unmodified. Therefore, having shellcode with only alphanumeric instructions is sometimes necessary and often preferred.

An alphanumeric instruction is an instruction where each of the four bytes of the instruction is either an upper case or lower case letter, or a digit. In particular, the bit patterns of these bytes must always conform to the following constraints:

- The most significant bit, bit 7, must be set to 0.
- Bit 6 or 5 must be set to 1.
- If bit 5 is set to 1, but bit 6 is set to 0, then bit 4 must also be set to 1.

These constraints do not eliminate all non-alphanumeric characters, but they can be used as a rule of thumb to quickly dismiss most of the invalid bytes. Each instruction will have to be checked whether its bit pattern follows these conditions and under which circumstances.

It is worth emphasizing that these constraints are tough: only 0.34% of the 32-bit words consist of 4 alphanumerical bytes.

This section will discuss some of the difficulties of writing alphanumeric code. When we discuss the bits in a byte, we will maintain the definition as introduced above: the most significant bit in a byte is bit 7 and the least significant bit is bit 0. Bits 31 to 24 form the first byte of an ARM instruction, and bits 7 to 0 form the last byte.

The ARM processor (in its v6 incarnation) has 147 instructions. Most of these instructions cannot be used in alphanumeric code, because at least one of the four bytes of the instruction is not alphanumeric. In addition, we have also filtered out instructions that require a specific version of the ARM processor, in order to keep our work as broadly applicable as possible.

3.1.1 Registers

In alphanumeric code, not all instructions that take registers as operands can use any register for any operand. In particular, none of the data-processing instructions can take registers `r0` to `r2` and `r8` to `r15` as the destination register `Rd`. The reason is that the destination register is encoded in the four most significant bits of the third byte of an instruction. If these bits are set to the value 0, 1 or 2, this would generate a byte that is too small to be alphanumerical. If the bits are set to a value greater than 7, the resulting byte value will be too high.

If these registers cannot be set as the destination registers, this essentially means that any calculated value cannot be copied into one of these registers using the data-processing instructions. However, being able to set the contents of some of these registers is very important. As explained in Section 2.1, ARM uses registers `r0` to `r3` to transfer parameters to functions and system calls.

In addition, registers `r4` and `r6` can in some cases also generate non-alphanumeric characters. The only registers that can be used without restrictions are limited to `r3`, `r5` and `r7`. This means that we only have three registers that we can use freely throughout the program.

3.1.2 Conditional Execution

Because the condition code of an instruction is encoded in the most significant bits of the first byte of the instruction (bits 31-28), the value of the condition code has a direct impact on the alphanumeric properties of the instruction. As a result, only a limited set of condition codes can be used in alphanumeric shellcode. Table 3.1 shows the possible condition codes and their corresponding bit patterns.

Unfortunately, the condition code `AL`, which specifies that an instruction should always be executed, cannot be used. This means that all alphanumeric ARM instructions must be executed conditionally. From the 15 possible condition codes, only five can be used: `CC` (Carry clear), `MI` (Negative), `PL` (Positive), `VS` (Overflow) and `VC` (No overflow). This means that we can only execute instructions if the correct condition codes are set and that the conditions that can be used when attempting conditional control flow are limited.

Bit Pattern	Name	Description
0000	EQ	Equal
0001	NE	Not equal
0010	CS/HS	Carry set/unsigned higher or same
0011	CC/LO	Carry clear/unsigned lower
0100	MI	Minus/negative
0101	PL	Plus/positive or zero
0110	VS	Overflow
0111	VC	No overflow
1000	HI	Unsigned higher
1001	LS	Unsigned lower or same
1010	GE	Signed greater than or equal
1011	LT	Signed less than
1100	GT	Signed greater than
1101	LE	Signed less than or equal
1110	AL	Always (unconditional)
1111	(used for other purposes)	

Table 3.1: The different condition codes of an ARM processor.

3.1.3 The Instruction List

Appendix A contains a table with the full list of ARMv6 instructions and whether they are suitable for alphanumeric shellcode or not. From this list of 147 instructions, we will now remove all instructions that require a specific ARM architecture version and all the instructions that we have disqualified based on whether or not they have bit patterns which are incompatible with alphanumeric characters.

For each instruction, the ARM reference manual specifies which bits *must be* set to 0 or 1, and which bits *should be* set to 0 or 1 (defined as SBZ or SBO in the manual). However, on our test processor if we set a bit marked as "should be" to something else, the processor throws an undefined instruction exception. In our discussion, we have treated "should be" and "must be" as equivalent, but we still note the difference in the instruction list in Appendix A in case this behavior is different on other processors (since this would enable the use of many more instructions).

Removing the incompatible instructions leaves us with 18 instructions: `B/BL`, `CDP`, `EOR`, `LDC`, `LDM(1)`, `LDM(2)`, `LDR`, `LDRB`, `LDRBT`, `LDRT`, `MCR`, `MRC`, `RSB`, `STM(2)`, `STRB`, `STRBT`, `SUB`, `SWI`.

Even though they can be used alphanumerically, some of the instructions have no or only limited use in the context of shellcode:

- **B/BL** the branch instruction uses the last 24 bits as an offset to the program counter to calculate the destination of the jump. After making these bits alphanumeric, the instruction would have to jump at least 12MB[1] from the current location, far beyond the scope of our shellcode.
- **CDP** is used to tell the coprocessor to do some kind of data processing. Since we cannot know which coprocessors may be available or not on a specific platform, we discard this instruction as well.
- **LDC** the load coprocessor instruction loads data from consecutive memory addresses into a coprocessor.
- **MCR/MRC** move coprocessor registers to and from ARM registers. While this instruction could be useful for caching purposes (more on this later), it is a privileged instruction before ARMv6.

The remaining thirteen instructions can be categorized in groups that contain instructions with the same basic functionality but that only differ in the details. For instance, `LDR` loads a word from memory into a register whereas `LDRB` loads a byte into the least significant bytes of a register. Even though these are two different instructions, they perform essentially the same operation.

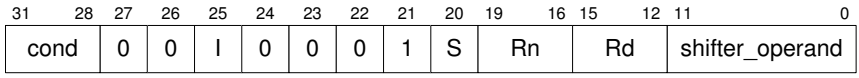
We can distinguish the following seven categories:

- **EOR** Exclusive OR
- **LDM (LDM(1), LDM(2))** Load multiple registers from a consecutive memory locations
- **LDR (LDR, LDRB, LDRBT, LDRT)** Load a value from memory into a register
- **STM** Store multiple registers to consecutive memory locations
- **STRB (STRB, STRBT)** Store a register to memory
- **SUB (SUB, RSB)** Subtract
- **SWI** Software Interrupt a.k.a. do a system call

Unfortunately, the instructions in the list above are not always alphanumeric. Depending on which operands are used, these functions may still generate characters that are non-alphanumeric. Hence, additional constraints apply to each instruction. In the following paragraphs, we discuss these constraints for the different instructions in the groups.

[1]The branch instruction will first shift the 24 bit offset left twice because all instructions start on a 4 byte boundary. This means that the smallest possible value we can provide as offset (0x303030) will in fact be an offset of 12632256.

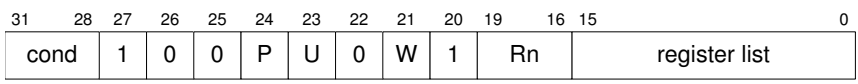
EOR



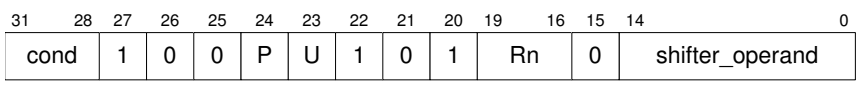

```
EOR{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
```

In order for the second byte to be alphanumeric, the S bit must be set to 1. If this bit is set to 0, the resulting value would be less than 47, which is not alphanumeric. *Rn* can also not be a register higher than `r9`. Since *Rd* is encoded in the first four bits of the third byte, it may not start with a 1. This means that only the low registers can be used. In addition, register `r0` to `r2` can not be used, because this would generate a byte that is too low to be alphanumeric. The shifter operand must be tweaked, such that its most significant four bytes generate valid alphanumeric characters in combination with *Rd*. The eight least significant bits are, of course, also significant as they fully determine the fourth byte of the instruction. Details about the shifter operand can be found in the ARM architecture reference manual [63].

LDM(1), LDM(2)



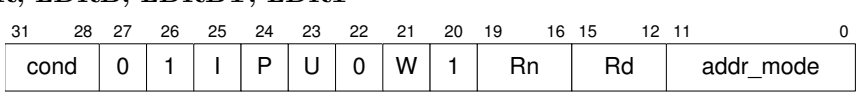

```
LDM{<cond>}<addressing_mode> <Rn>{!}, <registers>
```



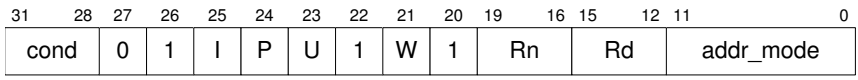

```
LDM{<cond>}<addressing_mode> <Rn>,
       <registers_without_pc>^
```

The list of registers that is loaded into memory is stored in the last two bytes of the instructions. As a result, not any list of registers can be used. In particular, for the low registers, `r7` can never be used. `r6` or `r5` must be used, and if `r6` is not used, `r4` must be used. The same goes for the high registers. Additionally, the U bit must be set to 0 and the W bit to 1, to ensure that the second byte of the instruction is alphanumeric. For *Rn*, registers `r0` to `r9` can be used with `LDM(1)`, and `r0` to `r10` can be used with `LDM(2)`.

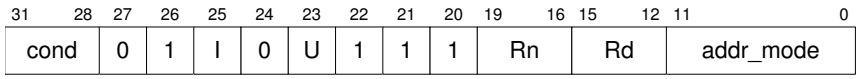
LDR, LDRB, LDRBT, LDRT



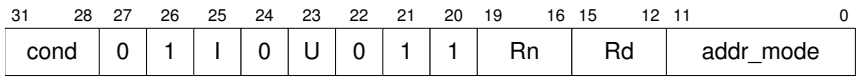
```
LDR{<cond>} <Rd>, <addressing_mode>
```




```
LDR{<cond>}B <Rd>, <addressing_mode>
```



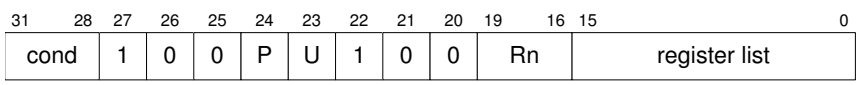

```
LDR{<cond>}BT <Rd>, <post_indexed_addressing_mode>
```



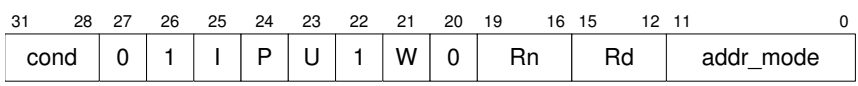

```
LDR{<cond>}T <Rd>, <post_indexed_addressing_mode>
```

The details of the addressing mode are described in the ARM reference manual and will not be repeated here for brevity's sake. However, the addressing mode must be specified in a way such that the fourth byte of the instruction is alphanumeric, and the least significant four bits of the third byte generate a valid character in combination with *Rd*. *Rd* cannot be one of the high registers, and cannot be `r0-r2`. The U bit must also be 0.

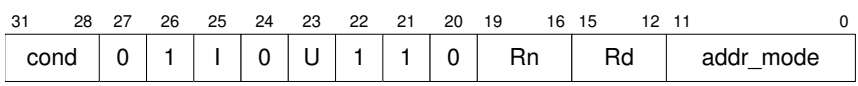
STM, STRB, STRBT




```
STM{<cond>}<addressing_mode> <Rn>, <registers>^
```



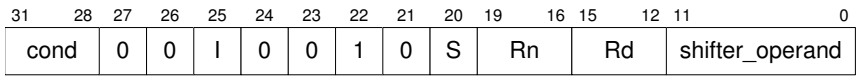

```
STR{<cond>}B <Rd>, <addressing_mode>
```



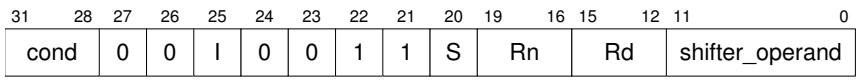

```
STR{<cond>}BT <Rd>, <post_indexed_addressing_mode>
```

The structure of `STM` is very similar to the structure of the `LDM` operation, and the structure of `STRB(T)` is very similar to `LDRB(T)`. Hence, comparable constraints apply. The only difference is that other values for *Rn* must be used in order to generate an alphanumeric character for the third byte of the instruction.

SUB, RSB



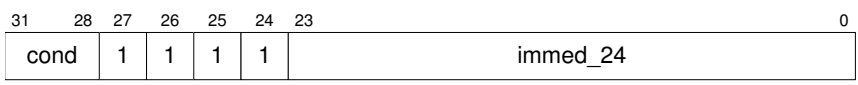

```
SUB{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
```




```
RSB{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
```

To get the second byte of the instruction to be alphanumeric, *Rn* and the S bit must be set accordingly. In addition, *Rd* cannot be one of the high registers, or `r0`-`r2`. As with the previous instructions, we refer to the ARM architecture reference manual [63] for a detailed instruction of the shifter operand.

SWI




```
SWI{<cond>} <immed_24>
```

As will become clear further in this chapter, it was essential for us that the first byte of the `SWI` call is alphanumeric. Fortunately, this can be accomplished by using one of the condition codes discussed in the previous section. The other three bytes are fully determined by the immediate value that is passed as the operand of the `SWI` instruction.

3.1.4 Self-modifying Code

One interesting way to get around the constraints of alphanumeric code, is to generate non-alphanumeric code using alphanumeric instructions, and then execute this non-alphanumeric code. The shellcode can contain a number of dummy

instructions, which can then be overwritten by other instructions to form non-alphanumeric instructions. Code that changes instructions at runtime is called *self-modifying code*.

ARM processors have an instruction cache, which makes writing self-modifying code a hard thing to do since all the instructions that are being executed will most likely already have been cached. The Intel x86 architecture has a specific requirement to be compatible with self-modifying code, and as such will make sure that when code is modified in memory the cache that possibly contains those instructions is invalidated. ARM has no such requirement, meaning that the instructions that have been modified in memory could be different from the instructions that are actually executed. Given the size of the instruction cache and the proximity of the modified instructions, it is very hard to write self-modifying shellcode without having to flush the instruction cache. We discuss how to do this in section 3.2.7

3.2 Writing Shellcode

In the previous section, we have sketched some of the problems that arise when writing alphanumeric code. However, there still are some problems that are specifically associated with writing shellcode. When the shellcode starts up, we know nothing about the program state, we do not know the value of any registers (including `CPSR`), the state of memory or anything else. This presents us with a number of important challenges to solve. This section will introduce a number of solutions for these problems. In addition, this section will show how to use the limited instructions that are available to simulate the operations of a much richer instruction set.

3.2.1 Conditional Execution

One way to make sure that a conditional instruction is executed, without knowing the value of the status register, is by duplicating instructions with mutually exclusive and also exhaustive condition codes. In our implementation, we have chosen the condition codes `PL` and `MI`. Instructions marked with `PL` will only be executed if the condition status is positive or zero. In contrast, `MI` instructions will only be executed if the condition status is negative. Clearly, this combination of condition codes is both mutually exclusive and exhaustive.

When our shellcode starts up, we can not be sure what state the `CPSR` register is in. Every instruction we want to be executed is added twice to the shellcode: a first time with a `PL` condition and a second time with a `MI` condition. Independent of what the contents of the status register is, exactly one of these instructions will be executed.

Once we gain more knowledge about the program state, we can execute an instruction that we know the result of, and mark it as an instruction that must update the `CPSR` register. This can be done, for example, by setting the S bit in a calculation with `SUB` or `EOR`. Setting the S bit on either instruction will still allow them to be represented alphanumerically. After these instructions are executed, the contents of the `CPSR` register is known, and the instructions do not have to be duplicated anymore.

3.2.2 Registers

When the processor starts executing the alphanumeric shellcode, the contents of all the registers is unknown. However, in order to do any useful calculations, the value of at least some registers must be known. In addition, a solution must be found to set the contents of registers `r0` to `r2`. Without these registers, the shellcode will not be able to do system calls or execute library functions.

Getting a constant in a register None of the traditional instructions are available to place a known value in a register, making this a non-trivial problem. The `MOV` instruction cannot be used, because it is never alphanumeric. The only data processing instructions that are available are `EOR` and `SUB`, but these instructions can only be used in conjunction with addressing modes that use immediate values or involve shifting and rotating. Because the result of a subtraction or exclusive OR between an unknown value and a known value is still unknown, these instructions are not useful. Given that these are the only arithmetic instructions that are supported in alphanumeric code, it is impossible to arithmetically get a known value into a register.

Fortunately, there is some knowledge about the running code that can be exploited in order to get a constant value into a register. Even though the exact value of the program counter, register `r15`, is unknown, it will always point to the executing shellcode. Hence, by using the program counter as an operand for the `LDRB` instruction, one of the bytes of the shellcode can be loaded into a register. This can be done as follows (non-alphanumeric code on the left, alphanumeric code on the right):

```
mov     r3, #...  |  subpl   r3, pc, #56
                  |  ldrplb  r3, [r3, #-48]
```

`pc` cannot be used directly in an `LDR` instruction as this would result in non-alphanumeric code. So its contents is copied to register `r3` by subtracting 56 from `pc`. The value 56 is chosen to make the instruction alphanumeric. Then,

register r3 is used in the LDRB instruction to load a known byte from the shellcode into r3. The immediate offset -48 is used to ensure that the LDRB instruction is alphanumeric. Once this is done, r3 can be used to load other values into other registers by subtracting an immediate value.

Loading values in arbitrary registers As explained in Section 3.1.1, it is not possible to use registers r0 to r2 as the destination registers of arithmetic operations. There is, however, one operation that can be used to write to the three lowest registers, without generating non-alphanumeric instructions. The LDM instruction loads values from the stack into multiple registers. It encodes the list of registers it needs to write to in the last two bytes of the instruction. If bit n is set, register Rn is included in the list and data is written to it. In order to get the bytes of the instruction to become alphanumeric, other registers have to be added to the list.

That is, the following code

```
mov r0, r3
mov r1, r4
mov r2, r6
```

has to be transformed as follows to be alphanumeric:

```
stmpldb r5, {r3, r4, r6, r8, r9, lr}^
rsbpl   r3, r8, #72
subpl   r5, r5, r3, ROR #2
ldmplda r5!, {r0, r1, r2, r6, r9, lr}
```

In the example above, the registers r3, r4 and r6 are stored on the stack using the STM instruction and then read from the stack into registers r0, r1, r2 using the LDM instruction. In order to make the STM instruction alphanumeric, the dummy registers r8, r9 and lr are added to the list, which will write them to the stack. Similarly the LDM instruction adds r6, r9 and lr. This will replace the value of r6 with the value of r8. The caret symbol is also necessary to make the instruction alphanumerical. This symbol sets a bit that is only used if the processor is executing in privileged mode. Setting the caret indicated that user mode registers should be stored instead of privileged mode registers. In unprivileged mode, the bit is ignored.

The *decrement before* addressing mode that is used for the STM instruction results in an invalid bit pattern when used in conjunction with LDM. Hence, we use a different addressing mode for the STM instruction. This requires, however, that we modify the starting address slightly for it to work as expected, which we do by subtracting 4 from the base register r5 using the RSB and SUB instructions above. Register r8 is assumed to contain the value 56 (for instance, by loading this value into the register as described in the previous paragraph). The RSB instruction will

subtract the contents of `r8` from the value 72, and store the result, 16, into `r3`. In the next instruction, `r3` is rotated right by two positions, producing the value 4.

3.2.3 Arithmetic Operations

The `ADD` instruction is not alphanumeric, so it must be simulated using other instructions. After generating a negative number by subtracting from our known value, an addition can be performed by subtracting that negative value from another register. However, one caveat is that when the `SUB` instruction is used with two registers as operands, an additional rotate right (`ROR`) on the second operand must be done in order to make the bytes alphanumeric. This effect can be countered by either rotating the second operand with an immediate value that will result in a (different) known value, or by rotating the second operand with a register that contains the value 0. The non-alphanumeric and alphanumeric variations of these operations are shown below.

```
mov     r7 , #-1        |  subpl    r7 , r3 , #57
mov     r3 , #0         |  subpl    r3 , r3 , #56
add     r5 , r5 , #1    |  subpl    r5 , r5 , r7 ROR r3
```

If we assume that register `r3` contains the value 56, using the trick explained in Section 3.2.2, the code above starts by setting register `r7` to -1 and sets register `r3` to 0. One is added to the value in register `r5` by subtracting the value -1 from it and rotating this value by 0 bits.

Subtract works in a similar fashion except a positive value is used as argument.

```
mov     r7 , #1         |  subpl    r7 , r3 , #55
mov     r3 , #0         |  subpl    r3 , r3 , #56
sub     r5 , r5 , #1    |  subpl    r5 , r5 , r7 ROR r3
```

The above examples show the $+1$ and -1 operations respectively. While these would be enough to calculate arbitrary values given enough applications, it is possible to use larger values by setting `r7` to a larger positive or negative value. However, for even larger values it is also possible to set `r3` to a nonzero value. For example, if `r3` is set to 20, then the last instruction will not subtract one, but will instead subtract 4096.

As can be seen from the example above, we can also subtract and add registers to and from each other (for addition, we of course need to subtract the register from 0 first).

Multiplication and division follow from repeated application of addition and subtraction.

3.2.4 Bitwise Operations

This section discusses the different bitwise operations.

Rotating and shifting Instructions on ARM that use the arithmetic addressing mode, explained in Section 2.1.4, can perform all kinds of shifts and rotations on the last operand prior to using it in a calculation. However, not all variants can be used in alphanumeric instructions. In particular, none of the left shift and left rotate variants can be used. Of course, left shifting can be emulated by multiplying by a power of 2, and left rotates can be emulated with right rotates.

Exclusive OR The representation of the Exclusive OR (`EOR`) instruction is alphanumeric and is thus one of the instructions that can be used in our shellcode. However the same restrictions apply as for subtract.

Complement By applying an Exclusive OR with the value -1 we can achieve a `NOT` operation.

Conjunction and disjunction Conjunction can be emulated as follows: for every bit of the two registers being conjoined, first shift both registers left[2] by 31 minus the location of the current bit, then shift the results to the right so the current bit becomes the least significant bit. We can now multiply the registers. We have now performed an `AND` over those bits. Shifting the result left by the amount of bits we shifted right will place the bit in the correct location. We can now add this result to the register that will contain the final result (this register is initialized to 0 before performing the `AND` operation). This is a rather complex operation, which turns out not to be necessary for proving Turing completeness or for implementing shell-spawning shellcode, but it can be useful if an attacker must perform an `AND` operation.

Given this implementation of `AND` and the previously discussed `NOT` operation, `OR` follows from the application of De Morgan's law.

3.2.5 Memory Access

Arbitrary values can be read from memory by using the `LDR` or `LDRB` instruction with a register which points 48 bytes further than the memory we wish to access:

[2]Left shifting is done by multiplying by the correct power of 2, as discussed in Section 3.2.3.

```
ldrpl    r5, [r3, #-48]!
ldrplb   r3, [r3, #-48]
```

The first instruction will load the four bytes stored at memory location `r3` minus 48 into `r5`. The offset calculation is written back into `r3` in order to make the instruction alphanumeric. The second instruction will load the byte pointed to by `r3` minus 48 into `r3`.

Storing bytes to memory can be done with the `STRB` instruction:

```
strplb   r5, [r3, #-48]
```

In the above example, `STRB` will store the least significant byte of `r5` at the memory location pointed to by `r3` minus 48.

The `STR` instruction cannot be used alphanumerically. An alternative to using `STR` is to use the `STM` instruction, which stores multiple registers to memory. This instruction stores the full contents of the registers to memory, but it cannot be used to store a single register to memory, as this would result in non-alphanumeric code.

Another possibility to store the entire register to memory is to use multiple STRB instructions and use the shift right capability that was discussed earlier to get each byte into the correct location

```
mov      r5, #0
mov      r3, #16
subpl    r3, r5, r7, ROR r3
subpl    r3, r5, r3, ROR r5
strplb   r3, [r13, #-50]
mov      r3, #24
subpl    r3, r5, r7, ROR r3
subpl    r3, r5, r3, ROR r5
strplb   r3, [r13, #-49]
```

The code above shows how to store the 2 most significant bytes of `r7`[3] to `r13` minus 49 and `r13` minus 50 respectively.

3.2.6 Control Flow

This section discusses unconditional and conditional branches.

[3]The code is slightly simplified for better readability in that we use `MOV`, which is not alphanumeric, to load the values to `r3` and `r5`

Unconditional branches As discussed in Section 3.1.3, the branch instruction requires a 24 bit offset from `pc` as argument, which is shifted two bits to the left and sign extended to a 32 bit value. The smallest alphanumeric offset that can be provided to branch corresponds to an offset of 12MB. In the context of shellcode, this offset is clearly not very useful. Instead, we will use self-modifying code to rewrite the argument to the branch before reaching this branching instruction. This is done by calculating each byte of the argument separately and using `STRB` with an offset to `pc` to overwrite the correct instruction. The non-alphanumeric (left) and alphanumeric (right) variations are shown below.

```
b   label
```

```
subpl   r3, pc, #48
subpl   r5, r8, #56
subpl   r7, r8, #108
subpl   r3, r3, r7, ROR r5
subpl   r3, r3, r7, ROR r5
subpl   r3, r3, r7, ROR r5

subpl   r7, r8, #54
strplb  r7, [r3, #-48]

.byte     0x30,0x30,0x30,0x90
```

The above example shows how the argument of a branch instruction can be overwritten. The branch instruction itself *is* alphanumeric, and is represented by byte `0x90` in machine code. In the example, the branch offset consists of three placeholder bytes with the value `0x30`. These will be overwritten by the preceding instructions.

The code copies `pc` minus 48 to `r3` and sets `r5` to 0 (we assume `r8` contains 56). It then sets `r7` to -52, subtracts this 3 times from `r3`. This will result in `r3` containing the value `pc` plus 108. When we subsequently write the value `r7` to `r3` minus 48 we will in effect be writing to `pc` plus 60. Using this technique we can rewrite the arguments to the branch instruction.

This must be done for every branch in the program before the branch is reached. However as discussed in section 3.1.4 we cannot simply write self-modifying code for ARM due to the instruction cache: this cache will prevent the processor from seeing our modifications. In section 3.2.7 we discuss how we were still able to flush the cache to allow our self-modifications to be seen by the processor once all branches have been rewritten.

Conditional branches In order to restrict the different types of instructions that should be rewritten, compare instructions and the corresponding conditional branch are replaced with a sequence of two branches that use only the `PL` and `MI` condition

codes. Some additional instructions must be added to simulate the conditional behavior that is expected.

As an example, imagine we want to execute the following instructions that will branch to the `endinter` label if `r5` is equal to 0:

```
cmp    r5, #0
beq    endinter
```

These two instructions can be rewritten as (`r8` contains 56):

```
subpl    r3, r8, #52
subpls   r3, r5, r3, ROR #2
bpl      notnull
subpl    r5, r8, #57
submis   r7, r8, #56
subpls   r5, r3, r5, ROR #2
bpl      endinter
submis   r7, r8, #56
notnull:
```

By observing whether the processor changes condition state after subtracting and adding one to the original value, we can deduce whether the original value was equal to zero or not. If we subtract one, and the state of the processor remains positive, the value must be greater than zero. If the processor changes state, the value was either zero or a negative number. By adding one again, and verifying that the processor state changes to positive again, we can ensure that the original value was indeed zero.

As with the unconditional branch, the actual branching instruction is not available in alphanumeric code, so again we must overwrite the actual branch instruction in the code above.

3.2.7 System Calls

As described in Section 3.1.4, the instruction cache of the ARM processor will hamper self-modifying code. One way of ensuring that this cache can be bypassed, is by turning it off programmatically. This can be done by using the alphanumeric `MRC` instruction, and specifying the correct operand that turns the cache off. However, as this instruction is privileged before ARMv6, we will not use this approach in our shellcode.

Another option is to execute a system call that flushes the cache. This can be done using the `SWI` instruction, given the correct operand. The first byte of a `SWI` instruction encodes the condition code and the opcode of the instruction. The other three bytes encode the number of the system call that needs to be executed.

Fortunately, the first byte can be made alphanumeric by choosing the `MI` condition code for the `SWI` instruction.

On ARM/Linux, the system call for a cache flush is 0x9F0002. None of these bytes are alphanumeric and since they are issued as part of an instruction this could mean that they cannot be rewritten with self-modifying code. However, `SWI` generates a software interrupt and to the interrupt handler 0x9F0002 is actually data. As a result, it will not be read via the instruction cache, so any modifications made to it prior to the `SWI` call will be reflected correctly, since these modifications will have been done via the data cache (any write or read to/from memory goes via the data cache, only instruction execution goes via the instruction cache).

In non-alphanumeric code, the instruction cache would be flushed with this sequence of operations:

```
mov  r0 , #0
mov  r1 , #-1
mov  r2 , #0
swi  0x9F0002
```

Since these instructions generate a number of non-alphanumeric characters, the previously mentioned code techniques will have to be applied to make this alphanumeric (i.e., writing to `r0` to `r2` via `LDM` and `STM` and rewriting the argument to `SWI` via self-modifying code). Given that the `SWI` instruction's argument is seen as data, overwriting the argument can be done via self-modification. If we also overwrite all the branches in the program prior to performing the `SWI`, then all self-modified code will now be seen correctly by the processor and our program can continue.

3.2.8 Thumb Mode

Although the Thumb instruction set is not used in order to prove that alphanumeric ARM code is Turing complete, it might nevertheless be interesting to know that it is possible to switch between the two modes in an alphanumeric way. Unlike the contents of the status register, the attacker typically knows beforehand whether the processor is in ARM mode or Thumb mode. If the attacker exploits a function that is running in ARM mode, the shellcode will be also be called in ARM mode. Likewise, an exploit for Thumb code will cause the shellcode to be started in Thumb mode.

Entering Thumb mode Changing the processor state from ARM mode to Thumb mode is done by calling the *branch and exchange* instruction `BX`. ARM instructions are always exactly four bytes and Thumb instructions are exactly two bytes. Hence, all instructions are aligned on either a two or four byte alignment. Consequently,

the least-significant bit of a code address will never be set in either mode. It is this bit that is used to indicate to which mode the processor must switch.

If the least significant bit of a code address is set, the processor will switch to Thumb mode, clear the bit and jump to the resulting address. If the bit is not set, the processor will switch to ARM mode. Below is an example that switches the processor from ARM to Thumb state.

```
subpl   r6, pc, #-1
bx      r6
<Thumb instructions>
```

In ARM mode, `pc` points to the address of the current instruction plus 8. The `BX` instruction is not alphanumeric, so it must be overwritten in order to execute the correct instruction. The techniques presented in Section 3.2.7 can be used to accomplish this.

Exiting Thumb mode If the program that is being exploited is running in Thumb mode when the vulnerability is triggered, the attacker can either choose to continue with shellcode that uses Thumb instructions, or he can switch to ARM mode. The `SWI` instruction is not alphanumeric in Thumb mode, making self-modifying code impossible with only Thumb instructions. The alternative is to switch to ARM mode, where system calls *can* be performed.

```
bx      pc
add     r7, #50
<ARM instructions>
```

Unlike ARM mode, the `BX` instruction *is* alphanumeric in Thumb mode. `pc` points to the address of the current instruction, plus 4. Since Thumb instructions are 2 bytes long, we must add a dummy instruction after the `BX` instruction. Also note that a dummy instruction before `BX` might be necessary in order to correct the Thumb alignment to ARM alignment.

3.3 Proving Turing-Completeness

In this section we argue that with our alphanumeric ARM shellcode we are able to perform all useful computations. We are going to show that the shellcode is *Turing complete*. Our argument runs as follows: we take a known Turing-complete programming language and build an interpreter for this language in alphanumeric shellcode.

The language of choice is BrainF*ck (`BF`) [73], which has been proven to be Turing complete [57]. `BF` is a very simple language that mimics the behavior of a Turing

machine. It assumes that it has access to unlimited memory, and that the memory is initialized to zero at program start. It also has a pointer into this memory, which we call the memory pointer. The language supports 8 different operations, each symbolized by a single character. Table 3.3 describes the meaning of each character that is part of the `BF` alphabet and gives the equivalent meaning in C (assuming that p is the memory pointer of type char*).

Table 3.2: The BF language

BF	**Meaning**	**C equivalent**
>	increases the memory pointer to point to the next memory location.	p++;
<	decreases the memory pointer to point to the previous memory location.	p--;
+	increases the value of the memory location that the memory pointer is pointing to by one.	(*p)++;
-	decreases the value of the memory location that the memory pointer is pointing to by one.	(*p)--;
.	write the memory location that the memory pointer is pointing to stdout.	write(1, p, 1);
,	read from stdin and store the value in the memory location that the pointer is pointing to.	read(0, p, 1);
[	starts a loop if the memory pointed to by the memory pointer is not 0. If it is 0, execution continues after the matching] (the loop operator allows for nested loops).	while (*p) {
]	continue the loop if the memory pointed to by the memory pointer is not 0, if it is 0, execution continues after the].	if (!*p) break; }

We implemented a mapping of `BF` to alphanumeric shellcode as an interpreter written in alphanumeric ARM shellcode. The interpreter takes as input any `BF` program and simulates the behavior of this program. The details of the interpreter are discussed below.

Several issues had to be addressed in our implementation.

- Because we wanted the `BF` program that must be executed to be part of the interpreter shellcode, we remapped all `BF` operations to alphanumeric characters: > ...] are mapped to the characters J ... C respectively.
- We extended the `BF` language (since this is a superset of BF, it is still Turing complete), with a character to do program termination. We use the character

"B" for this purpose. While this is not necessary to show Turing completeness, having a termination character simplifies our implementation.

- As with `BF` we assume that we have unlimited memory, our implementation provides for an initial memory area of 1024 bytes but this can be increased as needed.
- The memory required by our interpreter to run the `BF` program is initialized to 0 at startup of the interpreter.

3.3.1 Initialization

To support the `BF` language, we use three areas of memory: one which contains the code of the `BF` program (we will refer to this as the *BF-code* area) that we are executing, a second which serves as the memory of the program (the *BF-memory* area), and a third which we use as a stack to support nested loops (the *loop-memory* area). Memory for these areas is assumed to be part of the shellcode and each area is assumed to be 1024 bytes large.

We store pointers to each of these memory areas in registers `r10`, `r9` and `r11` respectively. These pointers are calculated by subtracting from the `pc` register. Once these registers are initialized, the contents of `BF`-memory is initialized to 0. Since it is part of our shellcode, the `BF`-memory contains only alphanumeric characters by default. The memory is cleared by looping (using a conditional branch) over the value of `r9` and setting each memory location to 0 until it reaches the end of the buffer. The memory size can be increased by adding more bytes to the `BF`-memory region in the shellcode, and by making minor modifications to the initialization of the registers `r9` to `r11`.

3.3.2 Parsing

Parsing the `BF` program is done by taking the current character and executing the expected behavior. To simplify the transition of the control flow from the code that is interpreting each `BF` code character to the actual implementation of the function, we use a jump table. The implementation of every `BF` operation is assumed to start 256 bytes from the other. By subtracting 'A' from the character we are interpreting and then subtracting that number multiplied by 256 from the program counter, we generate the address that contains the start of the implementation of that operation. To be able to end the program correctly we need the program termination character that was added to the `BF` language earlier ("B"). Because the implementation of a `BF` operation must be exactly 256 bytes, the actual implementation code is padded with dummy instructions.

3.3.3 BF Operations

The first four `BF` operations: “>”,“<”,“+” and “-” (or “J”, “I”, “H”, “G”) are easily implemented using the code discussed in Section 3.2. The instructions for “.” and “,” (“F” and “E”) are system calls to respectively read and write. As was discussed in Section 3.2.7, we need to rewrite the argument of the `SWI` instruction to correspond with the arguments for read and write (`0x00900004` and `0x00900003`), which can not be represented alphanumerically.

Loops in `BF` work in the following way: everything between “[” and “]” is executed repeatedly until the contents of the memory that the memory pointer is pointing to is equal to 0 when reaching either character. This scheme allows for nested loops. To implement these nested loops, we store the current pointer to the `BF`-code memory (contained in register `r10`) in the loop-memory area. Register `r11` acts as a stack pointer into this area. When a new loop is started, `r11` will point to the top of the stack. When we reach “]”, we compare the memory pointed to by the memory pointer to 0. If the loop continues, a recursive function call is made to the interpreted function. If the result was in fact 0, then the loop has ended and we can remove the top value of the loop-memory by modifying the `r11` register.

3.3.4 Branches and System Calls

As discussed in Section 3.2.6, we can not use branches directly: the argument for the branch instruction is a 24 bit offset from PC. Instead of overwriting the argument, however, we chose to instead calculate the address we would need to jump to and store the result in a register. We then insert a dummy instruction that will later be overwritten with the `BX <register>` instruction. Each possible branch instruction is fixed up in this way: at the end of a `BF` operation when we must jump to the end of the function, for the branches used to implement the loop instructions, ...

As discussed in Section 3.2.7, the arguments to system calls also need to be overwritten. This is also done by our self-modifying code.

All this self-modification is done right after the shellcode has started executing. Once we have overwritten all necessary memory locations, a cache flush is performed, which ensures that the new instructions will be read correctly when the processor reaches them.

3.4 Related Work

Building regular shellcode for ARM exists for both Linux [45] and Windows [50]. To facilitate NULL-byte avoidance, self-modification is also discussed in [45]. However, because only the arguments to SWI are modified, no cache flush is needed in this case, simplifying the shellcode considerably.

Alphanumeric shellcode exists for the x86 architecture [92]. Due to the variable length instructions used on this architecture, it is easier to achieve alphanumeric shellcode because many more instructions can be used compared to ARM architectures (jumps, for instance, are no problem), and the code is also not cached. Eller [36] discusses an encoder that will encode instructions as ASCII characters, that when executed on an Intel processor will decode the original instructions and execute them.

In Shacham [98] and Buchanan [20], the authors describe how to use the instructions provided by libc on both Intel and RISC architectures to perform return-into-libc attacks that are also Turing complete. By returning to a memory location which performs the desired instruction and subsequently executes a return, attackers can string together a number of return-into-libc attacks which can execute arbitrary code. The addresses returned to in that approach, however, may not be alphanumeric, which can result in problems when confronted with filters that prevent the use of any type of value.

3.5 Summary

We discussed how an attacker can use purely alphanumeric characters to insert shellcode into the memory space of an application running on a RISC processor. Given the fact that all instructions on a 32-bit RISC architecture are 4 bytes large, this turns out to be a non-trivial task: only 0.34% of the 32 bit words consist of 4 alphanumeric characters. However, we show that even with these severe constraints, it is possible to build an interpreter for a Turing complete language, showing that this alphanumeric shellcode is Turing complete. While the fact that the alphanumeric shellcode is Turing complete means that any program written in another Turing complete language can be represented in alphanumeric shellcode, an attacker may opt to simplify the task of writing alphanumeric shellcode in ARM by building a stager in alphanumeric shellcode that decodes the real payload, which can then be written non-alphanumerically.

In [124], we present real-world alphanumeric ARM shellcode that executes a pre-existing executable, demonstrating the practical applicability of the shellcode. Using alphanumeric shellcode, an attacker can bypass filters that filter out non-alphanumeric characters, while still being able to inject code that can perform

arbitrary operations. It may also help an attacker in evading intrusion detection systems that try to detect the existence of shellcode in input coming from the network.

Chapter 4

Code Pointer Masking

A major goal of an attacker is to gain control of the computer that is being attacked. This can be accomplished by performing a so-called *code injection attack*. In this attack, the attacker abuses a bug in an application in such a way that he can divert the control flow of the application to run binary code that the attacker injected in the application's memory space. The most basic code injection attack is a stack-based buffer overflow, where the attacker modifies the return address of a function. By making the return address point to code he injected into the program's memory as data, he can force the program to execute any instructions with the privilege level of the program being attacked [4]. Following this classic attack technique, several other — more advanced — attack techniques have been developed, including heap-based buffer overflows, indirect pointer overwrites, and others. All these attacks eventually overwrite a *code pointer*, i.e. a memory location that contains an address that the processor will jump to during program execution. Examples of code pointers are return addresses on the stack and function pointers.

According to the NIST's National Vulnerability Database [75], 9.86% of the reported vulnerabilities are buffer overflows, preceded only by SQL injection attacks (16.54%) and XSS (14.37%). Although buffer overflows represent less than 10% of all attacks, they make up 17% of the vulnerabilities with a high severity rating.

Code injection attacks are often high-profile, as a number of large software companies can attest to. Apple has been fighting off hackers of the iPhone since it has first been exploited with a code injection vulnerability in one of the iPhone's libraries[1]. Google saw the security of its new sandboxed browser *Chrome* breached[2] because of a code injection attack. And a recent attack exploiting a code injection

[1] CVE-2006-3459, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3459
[2] CVE-2008-6994, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-6994

vulnerability in Microsoft's Internet Explorer[3] led to an international row between Google and the Chinese government. This clearly indicates that even with the current widely deployed countermeasures, code injection attacks are still a very important threat.

Code injection attacks are typically the result of memory management bugs in applications that are written in so-called *unsafe* languages. Due to the potential of exploiting this type of bugs and the severity of what an exploit can do, a large number of countermeasures have been proposed to alleviate or solve these problems. Section 2.2 introduced the widely deployed countermeasures, and offered a short overview of other existing countermeasures. They can be roughly categorized in either highly performant but providing partial protection, or providing more complete protection at a substantial performance cost.

In this chapter we present a new approach, called *Code Pointer Masking (CPM)*, for protecting against code injection attacks. CPM is very efficient and provides protection that is partly overlapping with but also complementary to the protection provided by existing efficient countermeasures.

By efficiently masking code pointers, CPM constrains the range of addresses that code pointers can point to. By setting these constraints in such a way that an attacker can never make the code pointer point to injected code, CPM prevents the attacker from taking over the computer (under the assumptions listed in Section 1.5). Contrary to other highly efficient countermeasures, CPM's security does not rely on secret data of any kind, and so cannot be bypassed if the attacker can read memory [62, 111].

CPM has been developed in the context of a bilateral project between the DistriNet research group and DOCOMO Euro-Labs. The contents of this chapter builds on the general introduction to the ARM architecture presented in Section 2.1 and the introduction to code injection attacks presented in Section2.2.

4.1 Design

Existing countermeasures that protect code pointers can be roughly divided into two classes. The first class of countermeasures makes it hard for an attacker to change specific code pointers. An example of this class of countermeasures is Multistack [126]. In the other class, the countermeasures allow an attacker to modify code pointers, but try to detect these changes before any harm can happen. Examples of such countermeasures are stack canaries [27, 29], pointer encryption [28] and CFI [1]. These countermeasures will be further explained in Section 4.5.

[3]CVE-2010-0249, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0249

This section introduces the *Code Pointer Masking (CPM)* countermeasure, located in a third category of countermeasures. The key idea behind this new class is not to prevent overwriting code pointers, or detect memory corruptions, but to make it hard or even impossible for an attacker to do something useful.

4.1.1 Masking the Return Address

The basic design of CPM is explained using an example where an attacker modifies the return address of a function. The return address is one of the most popular code pointers that is used in attacks to divert the control flow. Other code pointers are protected in a similar fashion.

Listing 4.1 shows the sequence of events in a normal function epilogue. First, the return address is retrieved from the stack and copied into a register. Then, the processor is instructed to jump to the address in the register. Using for instance a stack based buffer overflow, the attacker can overwrite the return address on the stack. Then, when the function executes its epilogue, the program will retrieve the modified address from the stack, store it into a register, and will jump to an attacker-controlled location in memory.

Listing 4.1: A normal function epilogue.

```
[get return address from stack]
[jump to this address]
```

CPM mitigates this attack by strictly enforcing the correct semantics of code pointers. That is: *code pointers should never point to anything that is not in the code section of the application.* By disallowing the processor to jump to arbitrary locations in memory, the attacker will not be able to jump to the injected shellcode.

Before an application jumps to a code pointer retrieved from memory, the pointer is first modified such that it cannot point to a memory location that falls outside of the code section. This operation is called *masking.* The events in Listing 4.1 are enhanced as shown in Listing 4.2.

Listing 4.2: A CPM function epilogue.

```
[get return address from stack]
[apply bitmask on address]
[jump to this masked address]
```

By applying a mask, CPM will selectively be able to set or unset specific bits in the return address. Hence, it is an efficient mechanism to limit the range of addresses

that are possible. Any bitwise operator (e.g. AND, OR, BIC (bit clear — AND NOT), ...) can be used to apply the mask on the return address. Which operator should be selected depends on how the layout of the program memory is defined. On Linux, using an AND or a BIC operator is sufficient.

Even though an application may still have buffer overflow vulnerabilities, it becomes much harder for the attacker to exploit them in a way that might be useful. If the attacker is able to modify the return address of the function, he is only able to jump to existing program code.

Example The following example illustrates address masking for the Linux operating system. It should be noted that on other operating systems, the countermeasure may need different masking operations than used here, but the concept remains the same on any system.

As shown in Figure 4.1, program data, heap and stack are located above the program code in memory. For illustrative purposes, the program code is assumed to range from 0x00000000 to 0x0000FFFF, thus stack and heap are located on memory addresses larger than 0x0000FFFF.

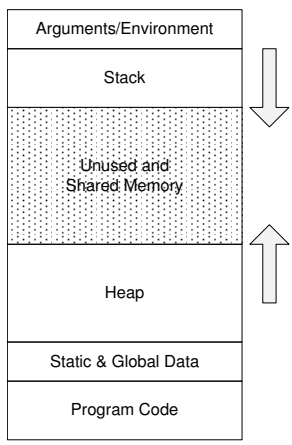


Figure 4.1: Stack, heap and program code memory layout for a Linux application

For each function in the application, the epilogue is changed from fetching the return address and jumping to it, to fetching the return address, performing an AND operation with the mask 0x0000FFFF on the return address, and then jumping to the result. Memory addresses that point to the stack or heap will have at least one bit of the two most significant bytes set. These bits will be cleared, however, because of the AND operation. As a result, before the memory address

reaches the JUMP instruction, it will be properly sanitized to ensure that it can only point to a location within the code segment.

The goal of CPM is to prevent code injection attacks. By overwriting the return address with rubbish, an attacker can still make the application crash, but he will not be able to execute his payload. However, the application will crash in an unpredictable way. Section 4.4 briefly discusses how CPM could be modified to detect attacks and abort the application instead of randomly crashing.

4.1.2 Mask Optimization

CPM, as described above, already protects against variations of return-to-libc attacks or return-oriented programming attacks where the attacker tries to jump to library code. Libraries are loaded outside of the application code region, thus returns from functions into library memory are not possible. However, an attacker is still able to jump to application code, including instructions that call library functions. This does not give the attacker the full freedom that he would typically enjoy, because he can only return to code that is present in the application or library methods that are used by the application. However, this may be enough to successfully exploit the application.

CPM solves this problem by calculating a specific mask per code pointer. If a function has only a few callers, these corresponding return addresses likely have some bits in common. These bits can be incorporated into the mask to make it more strict than the default mask. In addition, logic is added to the compiler to move methods around in order to generate better masks. Note that these masks are not secret. If an attacker knows the values of these masks, the only thing he can infer is the locations where he can jump to, which will in all likelihood be useless to him.

By also integrating a reordering phase in the compilation process, the compiler is able to shift the methods in such a way that the number of return sites that are allowed by the masks can be further optimized. The evaluation in Section 4.3.2 will show that it is possible to generate very narrow masks.

Example Assume that we have two methods M1 and M2, and that these methods are the only methods that call a third method M3. Method M3 can return to a location somewhere in M1 or M2. If we know during the compilation of the application that these return addresses are located at memory location 0x0B3E (0000101100111110) for method M1 and memory location 0x0A98 (0000101010011000) for method M2, we can compute a more restrictive mask for method M3. With the mask 0x0BBE (0000101110111110) we would do much better than with the default mask 0xFFFF (1111111111111111), since the more specific

mask 0x0BBE lets the method M3 return to M1 or M2, but not to the entire code memory region.

4.1.3 Masking Function Pointers

So far, we have only focused on the return address stored on the stack. For a more complete protection, all code pointers should be considered. If the application computes a code pointer somewhere and then jumps to it, this might lead to another vulnerability. By somehow influencing the computation, the attacker might be able to divert control flow. As a result, *all* computed jumps must be preceded with a masking operation.

It is very difficult to statically analyze a C program to know beforehand which potential addresses can be called from some specific function pointer call. CPM solves this by overestimating the mask it uses. During the compilation of the program, CPM knows the functions whose addresses are taken, and knows where all the function pointer calls are located. It changes the masks of the functions that are called to ensure that they can also return to the different return sites of the function pointer calls. In addition, the masks that are used to mask the function pointers are selected in such a way that they allow a jump to the different functions whose addresses have been taken somewhere in the program. As Section 4.3.1 shows, this has no important impact on the quality of the masks.

A potential issue is that calls of function pointers are typically implemented as a *JUMP <register>* instruction. There is a very small chance that if the attacker is able to overwrite the return address of a function and somehow influence the contents of this register, that he can put the address of his shellcode in the register and modify the return address to point to this *JUMP <register>* instruction. Even if this jump is preceded by a masking operation, the attacker can skip this operation by returning to the JUMP instruction directly. Although the chances for such an attack to work are extremely low (the attacker has to be able to return to the JUMP instruction, which will in all likelihood be prevented by CPM in the first place), CPM specifically adds protection to counter this threat.

The solutions to this problem differ from architecture to architecture. CPM can reserve a register that is used exclusively to perform the masking of code pointers. This will make sure that the attacker can never influence the contents of this register. The impact of this solution will differ from processor to processor, because it increases the register pressure. On architectures with only a small amount of registers, like the Intel x86 architecture, this solution might not be practical. However, as the performance evaluation in Section 4.3.1 shows, on the ARM architecture this *is* a good solution.

In the x86 prototype, this problem is solved by using segment selectors. The

masking of the return address is performed in a memory where the standard segment selector does not have access to. Hence, an attacker will never be able to jump directly to a CPM function return, because it will try to read the return address from a memory location that the attacker cannot influence.

4.1.4 Masking the Global Offset Table

A final class of code pointers that deserve special attention are entries in the *global offset table (GOT)*. The GOT is a table that is used for dynamically locating function calls in libraries. It contains the addresses of dynamically loaded functions that are located in libraries.

At program startup, these addresses are initialized to point to a helper method that loads the required library. After loading the library, the helper method modifies the addresses in the GOT to point to the library method directly. Hence, the second time the application tries to call a library function, it will jump immediately to the library without having to go through the helper method.

Overwriting entries in the GOT by means of indirect pointer overwriting is a common attack technique. By overwriting addresses in the GOT, an attacker can redirect the execution flow to his shellcode. When the application unsuspectedly calls the library function whose address is overwritten, the attacker's shellcode is executed instead.

Like the other code pointers, the pointers in the GOT are protected by masking them before they are used. Since all libraries are loaded into the 0x4NNNNNNN memory range, all code pointers in the GOT must either be somewhere in this memory range, or must point to the helper method (which lies in the program code memory).

4.1.5 Masking Other Code Pointers

CPM protects all code pointers in an application. This section contains the code pointers that have not been discussed yet, and gives a brief explanation of how they are protected.

When an application shuts down, it executes a number of so-called destructor methods. The *destructor table* is a table that contains pointers to these methods, making it a potential target for a code injection attack. If an attacker is able to overwrite one of these pointers, he might redirect it to injected code. This code will then be run when the program shuts down. CPM protects these pointers by modifying the routine that reads entries from the destructor table.

Applications also contain a *constructor table*. This is very similar to the destructor table, but runs methods at program startup instead of program shutdown. This table is not of interest to CPM, because the constructors will have already executed before an attacker can start attacking the application and the table is not further used.

The C standard also offers support for *long jumps*, a feature that is used infrequently. A programmer can save the current program state into memory, and then later jump back to this point. Since this memory structure contains the location of where the processor is executing, it is a potential attack target. CPM protects this code pointer by adding masking operations to the implementation of the longjmp method.

4.2 Implementation

The CPM prototypes are implemented in gcc-4.4.0 and binutils-2.20 for Linux on the ARM and x86 architectures. For GCC, the machine descriptions are changed to emit the masking operations during the conversion from RTL[4] to assembly. The ARM implementation provides the full CPM protection for return addresses, function pointers, GOT entries, and the other code pointers. The x86 implementation is a work in progress and currently only protects the return addresses.

4.2.1 Return Addresses

On Intel x86, the `RETURN` instruction is used to complete a function epilogue by popping the return address from the stack and jumping to it. Because the masking operation must happen between the pop from the stack and the jump, our prototype cannot use the `RETURN` instruction. Instead, the code is rewritten to the sequence of instructions in Listing 4.3. 0xNNNNNNNN represents the function-specific mask that is used. It is calculated by combining all the addresses where the function can return to using an OR operation.

Listing 4.3: A CPM function epilogue on x86.

```
popl    %edx
andl    $0xNNNNNNNN, %edx
jmp     *%edx
```

[4]RTL or *Register Transfer Language* is one of the intermediate representations that is used by GCC during the compilation process.

Function returns on ARM generally make use of the `LDM` instruction. `LDM`, an acronym for 'Load Multiple', is similar to a `POP` instruction on x86. But instead of only popping one value from the stack, `LDM` pops a variable number of values from the stack into multiple registers. In addition, the ARM architecture also supports writing directly to the program counter register. Hence, GCC uses a combination of these two features to produce an optimized epilogue. Listing 4.4 shows what this epilogue looks like.

Listing 4.4: A function prologue and epilogue on ARM.

```
stmfd   sp!, {<registers>, fp, lr}
...
ldmfd   sp!, {<registers>, fp, pc}
```

The `STMFD` instruction stores the given list of registers to the address that is pointed to by the `sp` register. *<registers>* is a function-specific list of registers that are modified during the function call and must be restored afterwards. In addition, the frame pointer and the link register (that contains the return address) are also stored on the stack. The exclamation mark after the `sp` register means that the address in the register will be updated after the instruction to reflect the new top of the stack. The 'FD' suffix of the instruction denotes in which order the registers are placed on the stack.

Similarly, the `LDMFD` instruction loads the original values of the registers back from the stack, but instead of restoring the `lr` register, the original value of this register is copied to `pc`. This causes the processor to jump to this address, and effectively returns to the parent function.

Listing 4.5: A CPM function prologue and epilogue on ARM.

```
stmfd   sp!, {<registers>, fp, lr}
...
ldmfd   sp!, {<registers>, fp}
ldr     r9, [sp], #4
bic     r9, r9, #0xNN000000
bic     r9, r9, #0xNN0000
bic     r9, r9, #0xNN00
bic     pc, r9, #0xNN
```

Listing 4.5 shows how CPM rewrites the function epilogue. The `LDMFD` instruction is modified to not pop the return address from the stack into `pc`. Instead, the return address is popped of the stack by the subsequent `LDR` instruction into the register `r9`. We specifically reserve register `r9` to perform all the masking operations of CPM. This ensures that an attacker will never be able to influence the contents of the register, as explained in Section 4.1.3.

Because ARM instructions cannot take 32-bit operands, we must perform the masking in multiple steps. Every bit-clear (`BIC`) operation takes an 8-bit operand, which can be shifted. Hence, four `BIC` instructions are needed to mask the entire 32-bit address. In the last `BIC` operation, the result is copied directly into `pc`, causing the processor to jump to this address.

The mask of a function is calculated in the same way as on the x86 architecture, with the exception that it is negated at the end of the calculation. This is necessary because the ARM implementation does not use the `AND` operator but the `BIC` operator.

Alternative function epilogues that do not use the `LDM` instruction are protected in a similar way. Masking is always done by performing four `BIC` instructions.

On the ARM architecture, the *long jumps* feature of C is implemented as an `STM` and an `LDM` instructions. The masking is performed in a similar way as explained here.

4.2.2 Function Pointers

The protection of function pointers is similar to the protection of the return address. Before jumping to the address stored in a function pointer, it is first masked with four `BIC` operations, to ensure the pointer has not been corrupted. Register `r9` is also used here to do the masking, which guarantees that an attacker cannot interfere with the masking, or jump over the masking operations.

The mask is calculated by statically analyzing the program code. In most cases, all possible function pointers are known at compilation time, and thus a specific mask can be generated by combining the different addresses using an `OR` operation. It is sometimes impossible to know the potential function pointers that might occur at runtime (for instance, when a library method returns a function pointer). In this case, the mask must be wide enough to allow for any potential code pointer (i.e. all function addresses in the program code, and all memory locations in the library memory region). It should be noted, however, that not a single application in the SPEC benchmark showed this behavior.

4.2.3 Global Offset Table Entries

As explained in Section 4.1.4, applications use a structure called *the global offset table* in order to enable dynamically loading libraries. However, an application does not interact directly with the GOT. It interacts with a jump table instead, called *the Procedure Linkage Table (PLT)*. The PLT consists of PLT entries, one for each library function that is called in the application. A PLT entry is a short

piece of code that loads the correct address of the library function from the GOT, and then jumps to it.

Listing 4.6: A PLT entry that does not perform masking.

```
add     ip, pc, #0xNN00000
add     ip, ip, #0xNN000
ldr     pc, [ip, #0xNNN]!
```

Listing 4.6 shows the standard PLT entry that is used by GCC on the ARM architecture. The address of the GOT entry that contains the address of the library function is calculated in the *ip* register. Then, in the last instruction, the address of the library function is loaded from the GOT into the `pc` register, causing the processor to jump to the function.

CPM protects addresses in the GOT by adding masking instructions to the PLT entries. Listing 4.7 shows the modified PLT entry.

Listing 4.7: A PLT entry that performs masking.

```
add     ip, pc, #0xNN00000
add     ip, ip, #0xNN000
ldr     r9, [ip, #0xNNN]!
cmp     r9, #0x10000
orrge   r9, r9, #0x40000000
bicge   pc, r9, #0xB0000000
bic     r9, r9, #0xNN000000
bic     r9, r9, #0xNN0000
bic     r9, r9, #0xNN00
bic     pc, r9, #0xNN
```

The first three instructions are very similar to the original code, with the exception that the address stored in the GOT is not loaded into `pc` but in `r9` instead. Then, the value in `r9` is compared to the value 0x10000.

If the library *has not* been loaded yet, the address in the GOT will point to the helper method that initializes libraries. Since this method is always located on a memory address below 0x10000, the `CMP` instruction will modify the status flags to 'lower than'. This will force the processor to skip the two following `ORRGE` and `BICGE` instructions, because the suffix 'GE' indicates that they should only be executed if the status flag is 'greater or equal'. The address in `r9` is subsequently masked by the four `BIC` instructions, and finally copied into `pc`.

If the library *has* been loaded, the address in the GOT will point to a method loaded in the 0x4NNNNNNN address space. Hence, the `CMP` instruction will set the status flag to 'greater than or equal', allowing the following `ORRGE` and `BICGE`

instructions to execute. These instructions will make sure that the most-significant four bits of the address are set to 0x4, making sure that the address will always point to the memory range that is allocated for libraries. The `BICGE` instruction copies the result into `pc`.

4.2.4 Limitations of the Prototype

In some cases, the CPM prototype cannot calculate the masks without additional input. The first case is when a function is allowed to return to library code. This happens when a library method receives a pointer to an application function as a parameter, and then calls this function. This function will return back to the library function that calls it.

The prototype compiler solves this by accepting a list of function names where the masking should not be done. Developers must take exceptional care when writing code for functions that will not be masked, or any child function. This list is program-specific and should be maintained by the developer of the application. In the SPEC benchmark, only one application has one method where masking should be avoided.

The second scenario is when an application generates code, and then tries to jump to it. This behavior can be necessary for applications such as just-in-time compilers. CPM will prevent the application from jumping to the generated code, because it is located outside the acceptable memory regions. A similar solution could be used as described in the previous paragraph. None of the applications in the SPEC benchmark displayed this behavior.

4.3 Evaluation

In this section, we report on the performance of our CPM prototype, and discuss the security guarantees that CPM provides.

4.3.1 Compatibility, Performance and Memory Overhead

To test the compatibility of our countermeasure and the performance overhead, we ran the SPEC benchmark [49] with our countermeasure and without. All tests were run on a single machine (ARMv7 Processor running at 800 MHz, 512 Mb RAM, running Ubuntu Linux with kernel 2.6.28).

All C programs in the SPEC CPU2000 Integer benchmark were used to perform these benchmarks. Table 4.1 contains the runtime in seconds when compiled with

SPEC CPU2000 Integer benchmarks				
Program	**GCC (s)**	**CPM (s)**	**Overhead**	**Mask size (bits)**
164.gzip	808	824	+1.98%	10.4
175.vpr	2129	2167	+1.78%	12.3
176.gcc	561	573	+2.13%	13.8
181.mcf	1293	1297	+0.31%	8.3
186.crafty	715	731	+2.24%	13.1
197.parser	1310	1411	+7.71%	10.7
253.perlbmk	809	855	+5.69%	13.2
254.gap	626	635	+1.44%	11.5
256.bzip2	870	893	+2.64%	10.9
300.twolf	2137	2157	+0.94%	12.9

Table 4.1: Benchmark results of the CPM countermeasure on the ARM architecture

the unmodified GCC on the ARM architecture, the runtime when compiled with the CPM countermeasure, and the percentage of overhead. There are no results for the Intel x86 prototype, because this is only a partial implementation of CPM. The results on the x86 architecture are, however, very much in line with the results on the ARM architecture.

Most applications have a performance hit that is less than a few percent, supporting our claim that CPM is a highly efficient countermeasure. There are no results for VORTEX, because it does not work on the ARM architecture. Running this application with an unmodified version of GCC results in a memory corruption (and crash).

The memory overhead of CPM is negligible. CPM increases the size of the binary image of the application slightly, because it adds a few instructions to every function in the application. CPM also does not allocate or use memory at runtime, resulting in a memory overhead of practically 0%.

The SPEC benchmark also shows that CPM is highly compatible with existing code. The programs in the benchmark add up to a total of more than 500,000 lines of C code. All programs were fully compatible with CPM, with the exception of only one application where a minor manual intervention was required (see Section 4.2.4).

4.3.2 Security Evaluation

As a first step in the evaluation of CPM, some field tests were performed with the prototype. Existing applications and libraries that contain vulnerabilities[5]

[5]CVE-2006-3459 and CVE-2009-0629

were compiled with the new countermeasure. CPM did not only stop the existing attacks, but it also raised the bar to further exploit these applications. However, even though this gives an indication of some of the qualities of CPM, it is not a complete security evaluation.

The security evaluation of CPM is split into two parts. In the first part, CPM is compared to the widely deployed countermeasures. Common attack scenarios are discussed, and an explanation is given of how CPM protects the application in each case. The second part of the security evaluation explains which security guarantees CPM provides, and makes the case for CPM by using the statistics we have gathered from the benchmarks.

CPM versus Widely Deployed Countermeasures

Table 4.2 shows CPM, compared in terms of security protection to the widely deployed countermeasures (see Section 2.2.2). The rows in the table represent the different vulnerabilities that allow code injection attacks, and the columns represent the different countermeasures.

Each cell in the table contains the different (combinations of) attack techniques (see Section 2.2.2) that can be used to break the security of the countermeasure(s). The different techniques that are listed in the table are *return-into-libc/return-oriented programming (**RiC**)*, *information leakage (**IL**)*, and *heap spraying (**HS**)*. As the table shows, CPM is the only countermeasure that protects against all different combinations of common attack techniques. Even the widely deployed combination of countermeasures can be bypassed with a combination of attack techniques.

Return-into-libc attacks, and the newer but related Return-oriented Programming attacks [98], are protected against by limiting the amount of return sites that the attacker can return to. Both attacks rely on the fact that the attacker can jump to certain interesting points in memory and abuse existing code (either in library code memory or application code memory). However, the CPM masks will most likely not give the attacker the freedom he needs to perform a successful attack. In particular, CPM will not allow returns to library code, and will only allow returns to a limited part of the application code.

Protection against heap spraying is easy for CPM: the masks will never allow an attacker to jump to the heap (or any other data structure, such as the stack). This makes heap spraying attacks by definition a non-issue for CPM.

CPM can also not leak information, because it uses no secret information. The masks that are calculated by the compiler are *not* secret. Even if an attacker knows the values of each individual mask (which he can, by simply compiling the same source code with the same CPM compiler), this will not aid him in circumventing the CPM masking process.

	ProPolice	ASLR[1]	No-Execute[2]	Combination[3]	CPM
Stack-based buffer overflow	IL	HS, IL	RiC	IL+RiC	/
Heap-based buffer overflow	N/A	HS, IL	RiC	IL+RiC, HS+RiC	/
Indirect pointer overwrite	N/A	HS, IL	RiC	IL+RiC, HS+RiC	/
Dangling pointer references	N/A	HS, IL	RiC	IL+RiC, HS+RiC	/
Format string vulnerabilities	N/A	HS, IL	RiC	IL+RiC, HS+RiC	/

[1] = This assumes the strongest form of ASLR, where the stack, the heap, and the libraries are randomized. On Linux, only the stack is randomized.
[2] = This assumes that all memory, except code and library memory, is marked as non-executable. On Linux, this depends from distribution to distribution, and is often not the case.
[3] = This is the combination of the ProPolice, ASLR and No-Execute countermeasures, as deployed in modern operating systems.

Table 4.2: An overview of how all the widely deployed countermeasures and CPM can be broken by combining different common attack techniques: Heap spraying (HS), Information leakage (IL) and Return-into-libc/Return-oriented programming (RiC).

Like many other compiler-based countermeasures, all libraries that an application uses must also be compiled with CPM. Otherwise, vulnerabilities in these libraries may still be exploited. However, CPM is fully compatible with unprotected libraries, thus providing support for linking with code for which the source may not be available.

CPM was designed to provide protection against the class of code injection attacks. This does not, however, exclude the other classes of attacks on C programs. In particular, data-only attacks [40], where an attacker overwrites only application data and no code pointers, are not protected against by CPM.

CPM Security Properties

The design of CPM depends on three facts that determine the security of the countermeasure.

CPM masks all code pointers. Code pointers that are not masked are still potential attack targets. For the ARM prototype, we mask all the different code pointers that are described in related papers. In addition, we looked at all the code that GCC uses to emit jumps, and verified whether it should be a target for CPM masking.

Masking is non-bypassable. All the masking instructions CPM emits are located in read-only program code. This guarantees that an attacker can never modify the instructions themselves. In addition, the attacker will not be able to skip the masking process. On the ARM architecture, we ensure this by reserving register *r9* and using this register to perform all the masking operations and the computed jumps.

The masks are narrow. How narrow the masks can be made differs from application to application and function to function. Functions with few callers will typically generate more narrow masks than functions with a lot of callers. The assumption that most functions have only a few callers is acknowledged by the statistics. In the applications of the SPEC benchmark, 27% of the functions had just one caller, and 55% of the functions had three callers or less. Around 1.20% of the functions had 20 or more callers. These functions are typically library functions such as *memcpy*, *strncpy*, ... To improve the masks, the compiler shuffles functions around and sprinkles a small amount of padding in-between the functions. This is to ensure that return addresses contain as many 0-bits as possible. With this technique, we can reduce the number of bits that are set to one in the different function-specific masks. Without CPM, an attacker can jump to any address in the memory (2^{32} possibilities). Using the techniques described here, the average

number of bits per mask for the applications in the SPEC benchmark can be brought down to less than 13 bits. This means that by using CPM for these applications, the average function is limited to returning to less than 0.0002% of the memory range of an application.

It is interesting to compare CPM to the Control-Flow Integrity (CFI, [1]) countermeasure. CFI determines a program's control flow graph beforehand and ensures that the program adheres to it. It does this by assigning a unique ID to each possible control flow destination of a control flow transfer. Before transferring control flow to such a destination, the ID of the destination is compared to the expected ID, and if they are equal, the program proceeds as normal. What makes CFI so interesting, is that it has been formally proven correct. Hence, under the assumptions made by the authors, an attacker will never be able to divert the control flow of an application that is protected with CFI.

If the masks can be made so precise that they only allow the correct return sites, an application protected with CPM will never be able to divert from the intended control flow. In this case, CPM offers the exact same guarantees that CFI offers. However, in practice, the masks will not be perfect. Hence, CPM can be seen as an efficient approximation of CFI.

The strength of protection that CPM offers against diversion of control flow depends on the precision of the masks. An attacker can still jump to any location allowed by the mask, and for some applications this might still allow interesting attacks. As such, CPM offers fewer guarantees than CFI. However, given the fact that the masks can be optimized, it is extremely unlikely that attackers will be able to exploit the small amount of room they have to maneuver. CPM also offers a performance that is up to 20x faster than CFI and —unlike CFI— supports dynamically linked code.

4.4 Discussion and Ongoing Work

CPM overlaps in part with other countermeasures, but also protects against attacks that are not covered. Vice versa, there are some attacks that might work on CPM (i.e. attacks that do not involve code injection, such as data-only attacks), which might not work with other countermeasures. Hence, CPM is complementary to existing security measures, and in particular can be combined with popular countermeasures such as stack canaries and ASLR. Adding CPM to the mix of existing protections significantly raises the bar for attackers wishing to perform a code injection attack. One particular advantage of CPM is that it is not vulnerable to a combination of different attack techniques, unlike the current combination of widely deployed countermeasures.

When an attacker overwrites a code pointer somewhere, CPM does not detect this modification. Instead it will mask the code pointer and jump to the sanitized address. An attacker could still crash the application by writing rubbish in the code pointer. The processor would jump to the masked rubbish address, and will very likely crash at some point. But most importantly, the attacker will not be able to execute his payload. CPM could be modified to detect any changes to the code pointer, and abort the application in that case. This functionality can be implemented in 7 ARM instructions (instead of 4 instructions), but does temporarily require a second register for the calculations.

A promising direction of future work is processor-specific enhancements. In particular, on the ARM processor, the conditional execution feature could be used to further narrow down the destination addresses that an attacker can use to return to. Conditional execution allows almost every instruction to be executed conditionally, depending on certain status bits. If these status bits are flipped when a return from a function occurs, and flipped again at the different (known) return sites in the application, the attacker is forced to jump to one of these return addresses, or else he will land on an instruction that will not be executed by the processor.

4.5 Related Work

Many countermeasures have been designed to protect against code injection attacks. In this section, we briefly highlight the differences between our approach and other approaches that protect programs against attacks on memory error vulnerabilities.

Bounds checkers. Bounds checking [9, 53, 55, 61, 79, 84, 108] is a better solution to buffer overflows, however when implemented for C, it has a severe impact on performance and may cause existing code to become incompatible with bounds checked code. Recent bounds checkers [2, 125] have improved performance somewhat, but still do not protect against dangling pointer vulnerabilities, format string vulnerabilities, and others.

Safe languages. Safe languages are languages where it is generally not possible for any known code injection vulnerability to exist as the language constructs prevent them from occurring. A number of safe languages are available that will prevent these kinds of implementation vulnerabilities entirely. There are safe languages [52, 58, 60, 77] that remain as close to C or C++ as possible, and are generally referred to as safe dialects of C. While some safe languages [26, 121] try to stay compatible with existing C programs, use of these languages may not always be practical for existing applications.

Probabilistic countermeasures. Many countermeasures make use of randomness when protecting against attacks. Many different approaches exist when using randomness for protection. Canary-based countermeasures [29, 42, 59, 93] use a secret random number that is stored before an important memory location: if the random number has changed after some operations have been performed, then an attack has been detected. Memory-obfuscation countermeasures [14, 28] encrypt (usually with XOR) important memory locations or other information using random numbers. Memory layout randomizers [13, 15, 114, 120] randomize the layout of memory: by loading the stack and heap at random addresses and by placing random gaps between objects. Instruction set randomizers [10, 54] encrypt the instructions while in memory and will decrypt them before execution.

While these approaches are often efficient, they rely on keeping memory locations secret. However, programs that contain buffer overflows could also contain "buffer overreads" (e.g. a string which is copied via *strncpy* but not explicitly null-terminated could leak information) [111] or other vulnerabilities like format string vulnerabilities, which allow attackers to print out memory locations. Such memory leaking vulnerabilities could allow attackers to bypass this type of countermeasure.

Separation and replication of information. Countermeasures that rely on separation or replication of information will try to replicate valuable control-flow information or will separate this information from regular data [23]. This makes it harder for an attacker to overwrite this information using an overflow. Some countermeasures will simply copy the return address from the stack to a separate stack and will compare it to or replace the return addresses on the regular stack before returning from a function [23]. These countermeasures can be bypassed using indirect pointer overwriting where an attacker overwrites a different memory location instead of the return address by using a pointer on the stack. More advanced techniques try to separate all control-flow data (like return addresses and pointers) from regular data [126], making it harder for an attacker to use an overflow to overwrite this type of data.

While these techniques can efficiently protect against buffer overflows that try to overwrite control-flow information, they do not protect against attacks where an attacker controls an integer that is used as an offset from a pointer.

Another widely deployed countermeasure distinguishes between memory that contains code and memory that contains data. Data memory is marked as non-executable [114]. This simple countermeasure is effective against direct code injection attacks (i.e. attacks where the attacker injects code as data), but provides no protection against indirect code injection attacks such as return-to-libc attacks. CPM can provide protection against both direct and indirect code injection.

Software Fault Isolation. Software Fault Isolation (SFI) [68, 118] was not

developed as a countermeasure against code injection attacks in C, but it does have some similarities with CPM. In SFI, data addresses are masked to ensure that untrusted code cannot (accidentally) modify parts of memory. CPM on the other hand masks code addresses to ensure that control flow can not jump to parts of memory.

Execution monitors. Some existing countermeasures monitor the execution of a program and prevent transferring control-flow which could be unsafe.

Program shepherding [56] is a technique that monitors the execution of a program and will disallow control-flow transfers[6] that are not considered safe. An example of a use for shepherding is to enforce return instructions to only return to the instruction after the call site. The proposed implementation of this countermeasure is done using a runtime binary interpreter. As a result, the performance impact of this countermeasure is significant for some programs, but acceptable for others.

Control-flow integrity, as discussed in Section 4.3.2, is also a countermeasure that is classified as an execution monitor.

4.6 Summary

We introduced Code Pointer Masking (CPM), an efficient mechanism to strongly mitigate the risk of code injection attacks in C programs. By masking code pointers before they are used, CPM imposes restrictions on these pointers that render them useless to attackers. CPM does not protect applications from data-only attacks, and, although the chances are very slim, it might still allow code injection attacks if the masks are not narrow enough.

CPM offers an excellent performance/security trade-off. It severely limits the risk of code injection attacks, at only a very small performance cost. It seems to be well-suited for handheld devices with slow processors and little memory, and can be combined with other countermeasures in a complementary way.

[6] Such a control flow transfer occurs when e.g., a *call* or *ret* instruction is executed.

Chapter 5

Security-by-Contract

One feature that modern security architectures, as discussed in Section 2.3, are still lacking is expressivity. The application either has a permission to perform an action, or does not have the permission. However, this means that users cannot express a desire to give an application permission to use a specific resource, *as long as it is not abused*. For instance, a user might want to give permission to a game to send short messages (if the game communicates via these messages with other players), but this does not mean that the user wants to give the game the ability to send an unbounded amount of messages to any phone number.

In addition, doing permission checks might be relatively cheap, but they are not free. An additional optimization would be to skip the permission check if the system could somehow verify beforehand that the application will never break the constraints that the user wants to impose.

This chapter describes the notion of *security-by-contract* (SxC, [35]). The SxC paradigm has been worked out in the European FP6 project *Security of Software and Services for Mobile Systems* (S3MS) by a consortium of five academic and seven industrial partners, under the supervision of Fabio Massacci. SxC supports a number of different technologies that verify whether applications adhere to a predefined security policy, and enforces this policy when no guarantees can be given about the program's behavior with respect to the policy. When a malicious application tries to access one of these restricted resources, the SxC framework will prevent it from doing so, under the assumptions specified in Section 1.5.

We then present our implementation and evaluation of the SxC paradigm on the Microsoft .NET Compact Framework. The different on-device and off-device components are discussed, together with the implementation strategy we used to build the framework. A number of practical implementation issues are highlighted

and discussed. The implementation is evaluated, and it is shown that our implementation performance characteristics are very good. The implementation has a similar granularity as the Android system in terms of interception points, but it allows for much more expressive policies to be written. In addition, there are mechanisms in place to remove redundant policy checks, speeding up the overall performance of the system.

5.1 General Overview

This section introduces the different concepts of Security-by-Contract, as worked out by the different partners in the S3MS consortium. All this work happened before the author of this PhD thesis was involved in the project.

Security-by-Contract is based on the premise that applications should only be trusted if it can be proven that they will not harm the system. In order to support such a system, a good description must be developed that clearly and unambiguously defines what is understood by 'harming the system', called a *system policy*. In addition, one or more mechanisms must be developed to somehow prove or enforce that the application will adhere to such a system policy. These mechanisms are called *enforcement technologies*, because they enforce that the application will never break the policy.

5.1.1 Policies and Contracts

Loosely speaking, a system policy is a set of rules to which an application must comply. These rules typically limit the access of an application to a specific part of the system API. For instance, there could be a set of rules to prohibit applications from accessing the network or to limit the access to the file system. These accesses are also called *security-related events* (SREs). One could think of a policy as an upper-bound description of what applications are allowed to do. System policies are defined by the owner of a mobile device.

Application contracts are very similar, but instead of defining the upper-bound of what an application *can* do, it describes the upper-bound of what the application *will* do. It is the 'worst case' scenario of security-related behavior of an application. The contract is typically designed by the application developer and is shipped together with the application as metadata.

To express the different system policies and application contracts, some kind of descriptive language is needed. The S3MS consortium defined two different policy languages, one that represents security automata by means of an explicit declaration of the security state and guarded commands that operate on this state, and another

one that is a variant of a temporal logic. Both languages extend history-based access control by introducing the notion of scopes. A scope specifies whether the policy applies to (1) a single run of each application, (2) saves information between multiple runs of the same application or (3) gathers events from the entire system. Even though they are called 'policy languages', these languages can also be used to specify application contracts.

CONSPEC

In the CONSPEC policy language [3], rules can be described on so-called security-related events (SRE). In the context of the S3MS implementations, these SREs correspond to security-sensitive calls from an application into its base class library. Examples of these calls are methods that open files, network connections or give access to sensitive data.

Listing 5.1: A CONSPEC policy, limiting the network data transfer

```
SCOPE Session
SECURITY STATE
    int bytesSent = 0;

BEFORE int sent = System.Net.Sockets.Socket.Send
      (byte[] array)
PERFORM
    array == null -> { }
    bytesSent + array.Length <= 1000 -> { }

AFTER int sent = System.Net.Sockets.Socket.Send
      (byte[] array)
PERFORM
    true -> { bytesSent += sent; }
```

Listing 5.1 contains a small example policy written in CONSPEC. The first line in the CONSPEC source sets the *scope* of the contract or policy. The scope defines whether the CONSPEC rules act on a single application instance, on all the instances of a specific application, or on every application in the system. A *session* scope acts on single application instances. Hence, the example of Figure 5.1 that imposes a 1000-byte network limit means that every instance of an application can send at most 1000 bytes. A second type of scope is a *global* scope. Rules in such a global scope act on all applications together. If we modify the example in Figure 5.1 to use a global scope instead of a session scope, the meaning of the rules would become "all applications on the system combined may send up to 1000 bytes over the network". Finally, a third *multi-session* scope is supported. This scope

fits in between the global and session scopes. A multi-session scope defines rules for all instances of the same application.

The scope declaration is followed by the security state. This security state contains a definition of all the variables that will be used to store the CONSPEC state. In the example of Figure 5.1, this will be a variable that holds the number of bytes that have already been sent. The ability for a CONSPEC policy to keep track of state is a critical difference between the SxC system and traditional security systems such as code access security. Where CAS can only either allow or deny a call to a specific library function, the SxC system can allow such a call as long as a particular precondition is met.

The security state is followed by one or more clauses. Each clause represents a rule on a security-relevant event. These rules can be evaluated before the SRE is called, after the SRE is called, or when an exception occurs. A clause definition consists of the 'BEFORE', 'AFTER' or 'EXCEPTIONAL' keyword to indicate when the rule should be evaluated, the signature of the SRE on which the rule is defined, and a list of guard/update blocks. The method signature corresponds largely to something a C# or Java programmer would expect.

As the name implies, a guard/update block consists of first a guard and then an update block. The guard is a boolean expression that is evaluated when a rule is being processed. If the guard evaluates to true, the corresponding update block is executed. All state changes that should occur can be incorporated in this update block. When a guard evaluates to true, the evaluation of the following guards (and consequently the potential execution of their corresponding update blocks) is skipped.

If none of the guards evaluates to true, this means the policy does not allow the SRE. For example, in Figure 5.1, if the current state of the policy has *bytesSent = 950*, then a call to the Send method with an array of length 55 will fail all the guards.

2D-LTL

The 2D-LTL policy language [32, 67] is an alternative for CONSPEC that is a temporal logic language based on a bi-dimensional model of execution. One dimension is a sequence of states of execution inside each run (session) of the application, and another one is formed by the global sequence of sessions themselves ordered by their start time.

The states of all the different applications are linked in two ways. For a single application, all its consecutive application states are linked to each other. This link is called *the session* of that application. In addition, all application sessions

are linked by a *frontier*. A frontier links all the latest active states of the different applications.

To reason about this bi-dimensional model, two types of temporal operators are applied: local and global ones. Local operators apply to the sequence of states inside the session, for instance, the "previously local" operator (YL) refers to the previous state in the same session, while "previously global" (YG) points to the final state of the previous session.

5.1.2 Enforcement Technologies

A number of different enforcement technologies can be used to make sure that an application complies with a device policy. This section gives an overview of the most common enforcement technologies.

Code Signing

One way to enforce a policy is to have a trusted third party certify that an application complies to a given policy. This trusted party would have a different public/private key pair for every different policy it certifies. An application developer would then send his application to this trusted party for compliance certification with one of the trusted party's policies. When the application is found to comply with the policy, it gets signed with the key corresponding to that policy. The signature can be contained in the application metadata, or can be embedded in the executable itself (using a mechanism such as 'Authenticode').

Notice the subtle difference between the meaning of the digital signature in the SxC system and in the standard Windows CE security architecture. Both systems make use of the exact same mechanism to verify the signature on an application, but on the SxC system, the signature tells more about the application than simply its origin. A signature in the SxC system certifies that the application will not violate a specific policy, whereas a signature in the Windows CE security architecture gives no precisely specified guarantees.

The upside of using this enforcement technology is that it is relatively simple to implement and use. However, third party certification can be costly, and requires trust in the certifying party.

Policy Matching

The operation of matching the application's claim with the platform policy is solved through *language inclusion* [24]. The contract and the policy are interpreted

as automata accepting sequences of SREs. Given two such automata Aut^C (representing the contract) and Aut^P (representing the policy), we have a match when the language accepted by Aut^C (i.e. the execution traces of the application) is a subset of the language accepted by Aut^P (i.e. the acceptable traces for the policy). For interesting classes of policies, the problem of language inclusion is decidable.

Proof-carrying Code

An alternative way to enforce a security policy is to statically verify that an application does not violate this policy. On the one hand, static verification has the benefit that there is no overhead at runtime. On the other hand, it often needs guidance from a developer (e.g. by means of annotations) and the techniques for performing the static verification (such as theorem proving) can be too heavy for mobile devices. Therefore, with proof-carrying code [76], the static verification produces a proof that the application satisfies a policy. In this way, the verification can be done by the developer, or by an expert in the field. The application is distributed together with the proof. Before allowing the execution of an application, a proof-checker verifies that the proof is correct for the application. Because proof-checking is usually much more efficient than making the proof, this step becomes feasible on mobile devices.

Inlining

A final enforcement technology is policy inlining. During the inlining process, the SxC system goes through the application code and looks for calls to SREs. When such a call is found, the system inserts calls to a monitoring component before and after the SRE. This monitoring component is a programmatic representation of the policy. It keeps track of the policy state and intervenes when an application is about to break the policy. After the inlining process, the application complies with a contract that is equivalent to the system policy.

The biggest advantage of this technique is that it can be used on applications that are deployed without a contract. It can be used as a fall-back mechanism when the other approaches fail and it can also ensure backwards compatibility. So, even in the case of contract-less applications, the SxC framework can offer guarantees that the application will not violate the system policy. A disadvantage is that the application is modified during the inlining process, which might lead to subtle bugs. Also, the monitoring of the SREs comes with a performance overhead.

5.1.3 Developing SxC Applications

To take full advantage of this new paradigm, applications have to be developed with SxC in mind. This means that some changes occur in the typical *Develop-Deploy-Run* application life cycle. Figure 5.1 shows an updated version of the application development life cycle.

Figure 5.1: The application development life cycle

The first step to develop an SxC compliant application, is to create a contract to which the application will adhere. Remember that the contract represents the security-related behavior of an application and specifies the upper-bound of calls made to SREs. Designing a contract requires intimate knowledge of the inner workings of the application, so it is typically done by a (lead-)developer or technical analyst. Some mobile phone operators, companies or other authorities may choose to publish contract templates that can then be used as a basis for new application contracts. Once the initial version of the contract has been specified, the application development can begin. During the development, the contract can be revised and changed when needed.

After the application development, the contract must somehow be linked to the application code in a tamper-proof way. One straightforward method to do this, is by having a trusted third party inspect the application source code and the contract. If they can guarantee that the application will not violate the contract, they sign a combined hash of the application and the contract. Another way to

link the contract and the code, is by generating a formal, verifiable proof that the application complies with the contract, and adding it to the application metadata container. When this step is completed, the application is ready to be deployed. The application is distributed together with its contract and optionally other metadata such as a digital signature from a third party or a proof.

When the program is deployed on a mobile device, the SxC framework checks whether the application contract is compatible with the device policy. It uses one of the different enforcement technologies, depending on the available metadata. For instance, if the application comes with a contract, the SxC framework may choose to use the *matching* enforcement technology [16]. If the application ships with a proof, the SxC framework may choose to try and prove compliance instead.

What happens is that the SxC framework checks whether the security behavior described in the contract is a subset of the security behavior allowed by the policy. If it is, the application is allowed to run as-is. If the contract is not a subset of the policy, the application is treated as an application without a contract.

Backwards Compatibility Support

The scenario in Section 5.1.3 showed how an application with SxC metadata would be produced and deployed to a mobile device. There is however an important need to also support the deployment of applications that are not developed with SxC in mind. This backwards compatibility is a make-or-break feature for the system.

When an application without a contract arrives on the mobile device, there is no possibility to check for policy compliance through matching. No metadata is associated with the application that can prove that it does not break the system policy. A solution for this problem is to enforce the system policy through runtime checking.

One example of a runtime policy enforcement technology is *inlining*. During the inlining process, the application is modified to intercept and monitor all the calls to SREs. When the monitor notices that the application is about to break the policy, the call to the SRE that causes the policy to be broken is canceled.

The strength of runtime checking is that it can be used to integrate non-SxC aware applications into the SxC process. A result of having this component in the SxC architecture is that it is usable as is, without having to update a huge amount of existing mobile applications.

Inlining can also be used when an existing application is made SxC-aware. It can sometimes be difficult to compose a contract for an application that may have been written years ago by a number of different developers. Instead of investing a lot of time (and money) into finding out which rules apply to the legacy application, an

inlining-based alternative could be a solution. The key idea is to use an inlining technique as described in Section 5.1.2. However, instead of inlining the application when it is loaded, the application is inlined by the developer. After being inlined, the application can be certified by a trusted third party.

There are a number of benefits of this approach, compared to on-device inlining. A first advantage is that the parts of the contract that can be manually verified by the trusted third party do not have to be inlined into the application. For instance, imagine that an existing application should comply with the policy *"An application cannot access the network, and cannot write more than 1000 bytes to the hard disk."* A third party can easily verify whether the application will break the first part of the policy by inspecting the program code. If the application contains no network-related code, it will not break this part of the policy. The second part of the policy may be harder to verify if the application does contain some file-related code but if it is unclear how many bytes are actually written to the hard disk. In this case, the developer could inline the application with a monitor that only enforces the second part of the policy, but the application will nevertheless be certified for the full policy.

Inlining large applications on a mobile device can be time consuming. Going through the inlining process before deploying the application eliminates this problem. It speeds up the application loading process, which is a second advantage.

A final advantage is that the developer can do a quality assurance check on the inlined application. This is useful to weed out subtle bugs and to ensure that the inlined application meets the expected quality standard of the developer.

5.2 S3MS.NET

This section presents an implementation of the Security-by-Contract paradigm on the .NET Compact Framework for Windows Mobile [31, 95]. All work presented in this section has been designed and implemented by the author of this PhD thesis, unless mentioned otherwise. An initial high-level architectural overview of the system was outlined by Lieven Desmet.

The S3MS.NET Framework can be divided into three parts. The first part is installed on a desktop computer, and is used to create and manage policies and the corresponding metadata. It is also used to deploy the S3MS.NET Framework and selected policies to mobile devices.

The second part lives on the mobile device. It takes care of verifying applications that are installed, and optionally modifies the original code to ensure that the application adheres to the system policy.

The third part also lives on the mobile device, and takes care of the runtime monitoring. If an application cannot prove that it will never break the policy, its security-relevant behavior is tracked by these components.

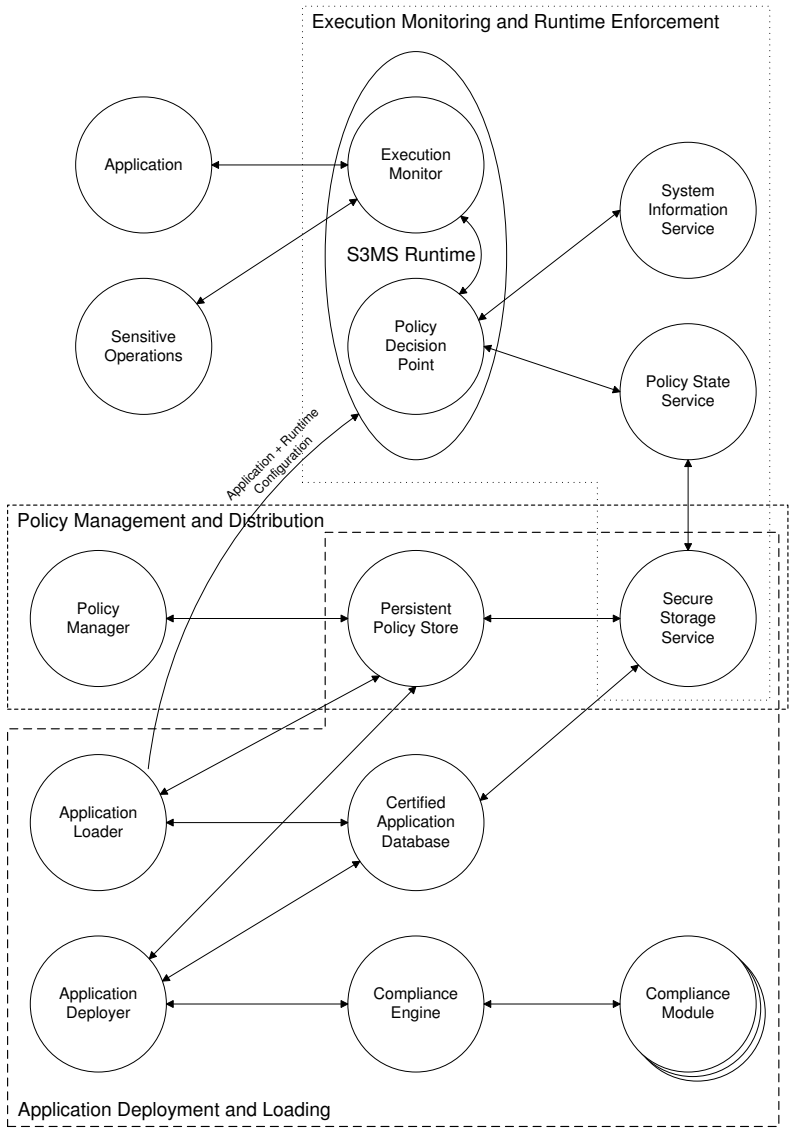


Figure 5.2: High-level architecture overview

Figure 5.2 shows a high-level architectural overview of the entire SxC system. Each of the three dotted lines corresponds to one of the three parts mentioned above.

Some components in the architecture are shared between different parts.

5.2.1 Policy Management and Distribution

System policies can be written by anyone, but in practice it can be expected that only a few people will actually write policies. Writing policies requires technical skills that are beyond those of most users of mobile devices. It can be anticipated that large companies or mobile phone operators will write policies centrally by their technical staff, and then distribute them to their employees' or customers' mobile devices. Policy writers can use the *Policy Manager* tool to write and edit policies, to prepare the policy for deployment, and to deploy it to a mobile device.

Preparing the policy for deployment means that the policy is converted from a textual policy to different formats that can easily be used by the on-device SxC framework. Multiple formats — also called *policy representations* — for one policy are possible, depending on which enforcement technologies the policy writer would like to support. Different enforcement technologies require different formats. For instance, if the policy has to support matching, a graph representation of the policy must be deployed to the mobile device. Likewise, if the policy writer wants to support inlining, an executable version of the policy must be generated. A collection of one or more policy representations is called a *policy package*.

The S3MS.NET implementation ships with a module to support the CONSPEC policy language, and a module, developed by Katsiaryna Naliuka and Fabio Massacci, to support the 2D-LTL policy language. The implementation also supports the code signing, matching and inlining enforcement technologies. It is built with a pluggable architecture to ensure that it can be extended further. For example, to add a new enforcement technology, a developer only has to build one class that implements the `IPolicyCompiler` class and register that class in the system. Figure 5.3 shows how textual representations of policies are compiled into different policy representations.

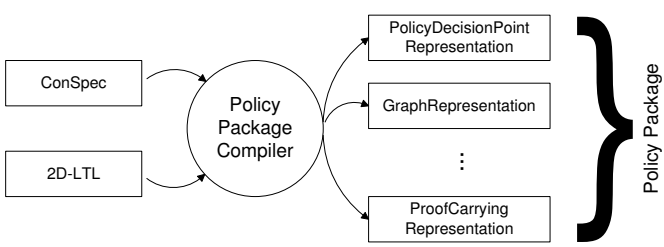


Figure 5.3: Compilation of a policy package

When the policy is ready, and the different policy representations are generated, a policy package is created that can be deployed to a mobile device. The Policy Manager sends the policy package to the *Persistent Policy Store*. This store is a container for all the policies that have been deployed on the device. Policies are saved on secure data storage, and can only be read by the Persistent Policy Store manager. The *Secure Storage Service* prohibits all access to the policies to other applications. If the device is equipped with a trusted platform module (TPM), this prohibition is enforced by the hardware.

5.2.2 Application Deployment and Loading

Applications that are deployed to a device can either be aware of the SxC framework or not. If an application is SxC-aware, it can be deployed with extra metadata that can be used by the SxC framework. Examples of such metadata are: a proof that the application complies with a specific policy, a digital signature of a trusted third party, or other data that is required for some policy enforcement technology.

When the *Application Loader* receives a request to execute an application, it sends the application to the *Application Deployer*. The deployer will first check whether the application was already verified before. In the architectural model, this is achieved by checking whether the application is present in the *Certified Application Database*. However, this database is a pure conceptual component. In the actual .NET implementation, the deployer checks whether the application is signed with a key that is managed by the SxC platform on the device. If an application is signed with this key, it has been processed before and it is considered to be compliant.

A non-compliant application is sent to the *Compliance Engine*. The purpose of this engine is to verify that the application adheres to the system policy, by trying every supported verification or enforcement method. These methods are represented by the different *Compliance Modules*. As soon as one of the compliance modules returns a result that indicates that the application conforms with the system policy, the compliance engine signs the application with the SxC key and executes the program. During this verification process, the actual contents of the executable may be changed. This is for instance the case with inlining. Figure 5.4 contains a graphical representation of this process.

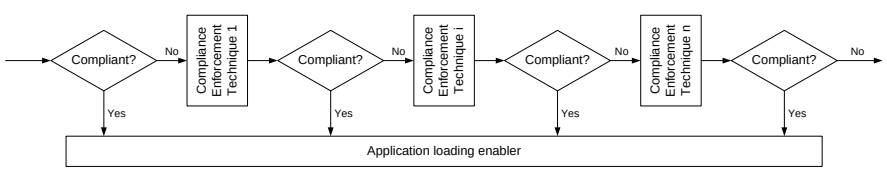


Figure 5.4: Verifying application/policy compliance

At deployment time of the policy, the policy writer can choose which policy enforcement mechanisms will be supported by the policy. Three enforcement mechanisms have been implemented in our prototype.

The first supported mechanism is the use of digital signatures set by a third party that certify compliance. This trusted party has a different public/private key pair for every different policy it certifies. If a mobile device administrator wishes to support a policy of this trusted third party, he configures a digital signature compliance module that is initialized with the public key that corresponds to this policy. When a signed application arrives on the device, this compliance module will check the signature against the public key of this third party. If the signature is valid, the application is allowed to run.

The second supported mechanism is matching, developed by Katsiaryna Naliuka and Fabio Massacci. At deployment time the target platform checks that the application security claims stated in the contract match with the platform policy. The matching procedure takes as input the application's contract and the mobile platform's policy in a suitable formal representation and then starts a depth first search procedure over the initial state. When a suspect state is reached we have two cases. First, when a suspect state contains an error state of the complemented policy then we report a security policy violation without further ado. Second, when a suspect state does not contain an error state of the complemented policy we start a new depth first search from the suspect state to determine whether it is in a cycle, i.e. it is reachable from itself. If it is we report an availability violation [34, 66].

The third supported enforcement mechanism is inlining for runtime monitoring [11, 39, 41, 48, 115], which is joint work between Dries Vanoverberghe and the author of this thesis. When an application arrives on a mobile device, and none of the compliance modules succeeds in verifying that the application does not violate the policy, the application is sent to the inlining module. The inlining module is also a compliance module, but it is always the last compliance module in the list. This is because the inlining process will never fail, but it does have the disadvantage that it imposes a runtime overhead, something that other enforcement techniques do not have. The bytecode of the application — in .NET, this is called the Intermediate Language (IL) code — is scanned for calls to security-relevant methods. Every call that is found is modified in such a way that before calling the method a call to a binary representation of the policy is made. The modified application can then be considered safe, because all calls to security-relevant functions will be intercepted and processed at runtime.

5.2.3 Execution Monitoring and Runtime Enforcement

The runtime enforcement scenario only comes into play when an application has been inlined. Other enforcement technologies are not active during the execution of

the program, because they can guarantee that the application will always comply with the policy before ever running the application. Runtime enforcement takes another approach, and lets applications execute without first formally proving (using either a mathematical proof, or a trust-based proof) that it will not violate the system policy. Instead, the application is instrumented with a monitoring library that can enforce the policy while the application is running.

The centerpiece of the execution monitoring implementation is the monitoring library — also called the *Policy Decision Point (PDP)*. This is the component where the policy logic is located. It interprets the current state and the requested action, and makes a decision to either allow or disallow the action. Calls from the application are received by the *Execution Monitor* and passed to the PDP. The PDP then requests the current state from the *Policy State Service* and may optionally request a number of system-specific settings from the *System Information Service*. The PDP can then execute the policy logic, update the state, and possibly disallow the call to the SRE. Figure 5.5 gives a schematic overview of this process.

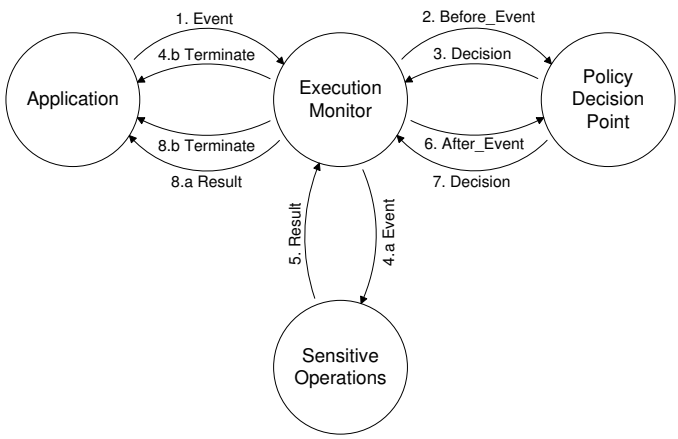


Figure 5.5: Execution monitoring

In our implementation, only the Execution Monitor is inlined. The Policy Decision Point is kept as a separate component that is called by the Execution Monitor. The inliner opens the executable file, and scans through it looking for security-related calls into the base class library. These calls correspond in our system with CONSPEC SREs. When such a call is found, the inliner inserts new instructions around this call. These instructions call into the policy decision point, which keeps track of the policy state. When, during runtime, the policy decision point notices that an application is going to break the policy, it intervenes and aborts the security-related call.

Identifying security-relevant methods statically in the presence of dynamic binding

and delegates (a form of type safe function pointers supported by the .NET virtual machine) is non-trivial. The tool implements the algorithm by Vanoverberghe and Piessens [116].

Listings 5.2 and 5.3 show the effect of inlining on a simple program that sends an SMS. If the method to send SMS's is considered security relevant, the inliner will transform it as shown. Note that the tool operates at the level of bytecode, not on source level, but we show the results as how they would look at source level to make the transformation easier to understand.

Listing 5.2: Example code that sends an SMS message on a mobile phone.

```
SmsMessage message = ...
message.SendSMS();
```

Listing 5.3: The SMS example code, after inlining.

```
SmsMessage message = ...
PDP.BeforeSendSMS(message);
try {
    message.SendSMS();
    PDP.AfterSendSMS(message);
} catch (SecurityException se) {
    throw se;
} catch (Exception e) {
    PDP.ExceptionSendSMS(message, e);
    throw;
}
```

Before each SRE call, a *'before handler'* is added, which checks whether the application is allowed to call that method. If not, an exception is thrown. This exception will prevent the application from calling the method, since the program will jump over the SRE to the first suitable exception handler it finds.

Likewise, after the SRE call, an *'after handler'* is added. This handler typically only updates the internal state of the PDP. If an exception occurs during the execution of the SRE, the *'exceptional handler'* will be called instead of the *'after handler'*. In summary, the different handler methods implement the reaction of the security automaton to the three types of events: calls, normal returns and exceptional returns of security-relevant methods.

Inheritance and Polymorphism

The simplified code shown above does not deal with inheritance and dynamic binding. Support for this was implemented by extending the logic in the PDP to consider the type of the object at runtime, instead of only looking at the static type that is available during the inlining process. When a security-relevant virtual method is called, calls are inlined to so-called *dynamic dispatcher methods* that inspect the runtime type of the object and forward to the correct handler. The details, and a formal proof of correctness of this inlining algorithm is presented in [116].

Multithreading and Synchronization

Inlining in a multithreaded program requires synchronization. Two synchronization strategies are possible: strong synchronization, where the security state is locked for the entire duration of a SRE call, or weak synchronization where the security state is locked only during execution of the handler methods.

Our tool implements strong synchronization, which might be problematic when SREs take a long time to execute, or are blocking (e.g. a method that waits for an incoming network connection). To alleviate this problem, the tool partitions the handler methods according to which security state variables they access. Two partitions that access a distinct set of state variables can be locked independently from each other.

5.3 Evaluation

The implementation of the S3MS.NET framework, as well as supporting documentation and examples can be found at:
`http://people.cs.kuleuven.be/~pieter.philippaerts/inliner/`

In the context of the S3MS project, we gained experience with the implementation described in this chapter on two case studies:

- a “Chess-by-SMS” application, where two players can play a game of chess on their mobile phones over SMS.
- a multiplayer online role-playing game where many players can interact in a virtual world through their mobile phones. The client for this application is a graphical interface to this virtual world, and the server implements the virtual world, and synchronizes the different players.

In addition, the implementation was used to support a project assignment for a course on secure software development at the K.U.Leuven. In this assignment, students were asked to enforce various policies on a .NET e-mail client.

Based on these experiences, we summarize the major advantages and limitations of the framework.

A major advantage of the framework, compared to state-of-the-art code access security systems based on sandboxing (such as .NET CAS and the Java Security Architecture) is its improved expressiveness. The main difference between CAS and the approach outlined here, is that CAS is stateless. This means that in a CAS policy, a method call is either allowed for an application or disallowed. With the S3MS approach, a more dynamic policy can be written, where a method can for instance be invoked only a particular number of times. This is essential for enforcing policies that specify quota on resource accesses.

A second important strength of the implementation is its performance. A key difference between CAS and our approach, is that CAS performs a stack walk whenever it tries to determine whether the application may invoke a specific sensitive function or not. Because stack walks are slow, this may be an issue on mobile devices (CAS is not implemented on the .NET Compact Framework). The speed of the S3MS approach mainly depends on the speed of the *before* and *after handlers*. These can be made arbitrarily complex, but typically require only a few simple calculations. This results in a small performance overhead. Microbenchmarks [117] show that the performance impact of the inlining itself is negligible, and for the policies and case studies done in S3MS, there was no noticeable impact on performance.

Finally, the support for multiple policy languages and multiple platforms makes the framework a very versatile security enforcement tool.

A limitation is that we do not support applications that use reflection. Using the reflection API, functions can be called dynamically at runtime. Hence, for security reasons, access to the reflection API should be forbidden, or the entire system becomes vulnerable. We do not see this as a major disadvantage, however, because our approach is aimed at mobile devices, and the reflection API is not implemented on the .NET Compact Framework. Also, by providing suitable policies for invoking the Reflection API, limited support for reflection could be provided.

A second limitation of the implemented approach is that it is hard and sometimes even impossible to express certain useful policies as security automata over API method calls. For instance, a policy that limits the number of bytes transmitted over a network needs to monitor all API method calls that could lead to network traffic, and should be able to predict how much bytes of traffic the method will consume. In the presence of DNS lookups, redirects and so forth, this can be very hard.

A final limitation is that the policy languages supported by the framework are targeted to "expert" users. Writing a correct policy is much like a programming task. However, more user-friendly (e.g. graphical) policy languages could be compiled to CONSPEC or 2D-LTL.

5.4 Related Work

The first widely deployed security architecture is based on trust-based signatures on applications. Applications that are deployed onto the mobile phone can be signed by a trusted third party, often called *Symbian signing* [64]. Depending on whether the system trusts the principal that signed the application, a decision is made to allow the application or not. The Windows CE security architecture [71] goes a step further and assigns different permission sets to applications, depending on their signature. If the signature is valid and trusted, the application is run in a fully trusted context. Otherwise, the application is either run in a partially trusted context, or not at all. The BlackBerry supports a security architecture that is very similar to the one present in Windows CE. Depending on the application's signature, it can get access to different sets of API functions. The Java Mobile Information Device Profile [113] takes a similar approach.

Android probably has the most advanced security model for mobile devices that is in widespread use today [5]. All applications run on top of the Dalvik virtual machine, and can be granted a set of permissions. These permissions are granted at install time, and cannot be changed later on. Different permissions give access to different parts of the Android API. Android uses application signatures to link different applications to a single developer, but does not use them to decide on the trustworthiness of an application. This architecture has been further enhanced in [37]. Ongtang et al. [80] added runtime policy enforcement for all IPC calls between Android applications. Every IPC call is intercepted and matched to a set of policies. If the policies expressly forbid the interaction, the IPC call is canceled.

5.5 Summary

We have argued that the classic security architecture of Windows CE is not well adapted to protect users from malicious roaming applications. In particular, the digital signatures used in Windows CE do not offer any guarantee of what the code will actually do. Section 5.1 introduced the notion of *Security-by-Contract (SxC)* and shows that this is superior to the Windows CE security architecture.

The SxC architecture can be used to secure untrusted roaming code, and protect users from malicious applications. By defining security contracts for applications

during their development, and by matching these contracts with device policies during deployment, compliance with the policy can be verified without incurring a runtime overhead. Legacy applications, or applications with contracts that cannot be verified, can be executed under the supervision of an execution monitor that will prevent the monitored application from violating the policy.

We have implemented the SxC paradigm on top of the Microsoft .NET Compact Framework for Windows Mobile devices. Our evaluation shows that using our framework imposes no or a very small runtime overhead, while actively protecting the system from abusive applications.

Chapter 6

Conclusion

We summarize this dissertation by revisiting the goals set forth in Section 1.3. The contributions are highlighted and linked to the different goals. We will then look at the ongoing work, related to the contributions of this dissertation. Finally, we conclude with a look at the future challenges in the field of mobile phone security.

6.1 Contributions

In Section 1.3, we set forth a number of goals for this dissertation. This section explains how these goals are met by the contributions.

We wanted to show that security is an important topic for mobile phones. An attacker that takes control over a mobile phone has unlimited access to all the resources of the phone. This is a serious threat, and users often underestimate the danger of this. We have proven that attackers can still mount a full and unrestricted attack on mobile phones, even in the presence of severe limitations on shellcode. The contributions in Chapter 3 show that if the attacker is limited to a small subset of all ARM instructions, consisting of all the instructions whose bytes are alphanumeric, he can still execute arbitrary code. This subset has been proven to be Turing Complete by implementing an interpreter for the BrainF*ck language, which is itself Turing Complete.

We wanted to build tools that can be used by phone manufacturers to secure their applications. The key here was to make sure that it is more difficult for an attacker to exploit a bug in the application. In other words, we wanted to make sure that bugs do not become vulnerabilities. Chapter 4 presented our solution to this problem. It introduced a new countermeasure against code injection attacks

called *Code Pointer Masking (CPM)*. This countermeasure is implemented as an extension of the GCC compiler. During the compilation process, the compiler emits additional code that checks at runtime whether code pointers point to the contents they are supposed to point to. If an attacker succeeds in modifying a code pointer, the CPM countermeasure will mask the modified code pointer, which makes sure that the attacker will not be able to jump to injected shellcode.

We wanted to minimize the potential risk of bugs in applications, and we also wanted the phone to be secure in the presence of malicious applications. These applications are installed and executed by the user, and thus are much harder to detect than other malware. Our solution is presented in Chapter 5, where we report on our implementation of the *Security-by-Contract (SxC)* paradigm. In this paradigm, additional checks ensure that an application will not violate a predetermined system policy. This policy will set restrictions on the usage of certain critical or costly resources of the mobile phone, effectively limiting the amount of damage that a malicious application can do. Our work focused on the implementation of the different tools and architecture, and on the inliner component. The inliner rewrites applications during deployment time, and embeds runtime checks that monitor the usage of different resources, and optionally blocks access to them.

Our tools and frameworks are highly performant and very compatible with existing code. The CPM countermeasure has a performance overhead of only a few percentage points, and a memory overhead of practically 0%. It is also very compatible with existing code bases, as was shown by the SPEC benchmark. Only one manual intervention had to be made for the entire 500,000+ lines-of-code benchmark to compile.

The SxC implementation shares the same characteristics. If an application that is being deployed can somehow prove that it will never break the system policy, the SxC framework will not impose any overhead at all. However, for other applications, a runtime monitor is embedded in the application. The performance of this monitor depends almost entirely on the complexity of the policy that is being enforced. Our SxC implementation is very compatible with existing code; not a single application has been reported to not work with the prototype.

In summary, it can be said that the various goals of this dissertation are met, as we have presented and implemented a number of different technological improvements upon the state of the art. However, not until these and other improvements have gone mainstream can we say that the state of mobile phone security is improving.

6.2 Ongoing Work

The work presented in this PhD thesis is still ongoing. Some related topics that could be worked out in future work are discussed in this section.

Alphanumeric Turing Complete Shellcode In Chapter 3 we have proven that alphanumeric shellcode is Turing Complete on ARM. We did this by implementing an interpreter for the BrainF*ck language. Although this is sufficient to prove Turing Completeness, it is difficult to actually use this interpreter in a real attack.

One interesting extension would be to write an instruction decoder in alphanumeric shellcode. The idea works as follows: the attacker writes some malicious code that he wants to execute. After compiling this code, it will most likely not be alphanumeric, and hence cannot be used as-is under the assumption that non-alphanumeric data is likely to get mingled. The attacker could, however, encode this program using an algorithm like the Base32 algorithm, which essentially converts this non-alphanumeric array of bytes to a (longer) alphanumeric array of bytes. If we have an alphanumeric decoder that decodes this encoded string and starts executing it, then we would have an interesting tool to deploy any malicious shellcode without adding a lot of complexity to the attacker.

Mason et al. [65] have shown that it is possible to generate shellcode for the x86 architecture that resembles written English text. This makes it even harder for intrusion detection systems to try and intercept shellcode by analyzing input data. It would be interesting to try a similar strategy for the ARM processor. This would probably be much more difficult than on the x86 architecture, but it would be interesting to see how far one could get with this.

Code Pointer Masking The CPM countermeasure is still in active development. A number of improvements are either being worked out, or are on the drawing board to be implemented at a later stage.

We already have a partial implementation of CPM on the Intel x86 architecture that protects the return address. This can be extended to offer full protection for all the remaining code pointers. Some architecture specific challenges will have to be overcome, but no major blocking issues are expected here.

CPM also does not protect dynamically loaded libraries. A system could be implemented where the binary code of libraries contains placeholder instructions at certain key locations, and where these placeholders are overwritten by masking instructions when the library is loaded into memory. This requires changes to the compiler, which must emit the required placeholder instructions, and also to the loader framework, which must rewrite the loaded libraries. In addition, it might

also be necessary that libraries and programs ship with some kind of metadata that contains information to calculate the different masks at runtime.

On the ARM architecture, there are also interesting hardware features that could be used in order to further increase the security guarantees of the countermeasure. In particular, the support for conditional execution on the ARM architecture can be used to further narrow down the different return sites an attacker can return to. New hardware features that are specifically aimed at improving application security could also be designed and implemented of course.

Finally, the algorithm that is currently used to shuffle around the different methods at compilation time in order to optimize masks is relatively straightforward. Different schemes could be implemented to see if the masks could be narrowed down even further.

Security-by-Contract A major source of ongoing work for the Security-by-Contract paradigm, is the development of tools that make writing policies and contracts easy. At this moment, writing policies and contracts is very difficult. Only technical people with an intimate knowledge of the framework for which the policies are being developed can write them. But even for a skilled policy writer, it can become very cumbersome to write or update large policies. Also, the larger the policy gets, the more likely it is that some holes are left uncovered. Some work has already been done to make this process easier [89, 104], but more work is required.

One of the main drawbacks of the current S3MS.NET implementation, is the fact that adding support for reflection to a policy is difficult. The main reason for this, is that we implemented the prototype to use client-side inlining instead of library-side runtime checking. This design choice was motivated by the fact that we do not have access to the source code of the .NET (Compact) Framework, which would be a prerequisite in order to build a library-side runtime monitor. It also has the additional benefit that there is no additional overhead when runtime monitoring is not required to guarantee the safety of an application. It would be interesting to build another prototype on one of the open source .NET frameworks (i.e. the Mono Framework [78] or the Rotor Framework [70]) that implements library-side runtime monitoring. The expected overhead is likely to be very small, and this would automatically enable support for reflection.

6.3 Future Challenges

Software firms are spending more time educating developers to write better and more secure code, because they realize that it is a huge competitive disadvantage to have buggy and exploitable applications. Large software companies have development

processes in place where security is one of the cornerstones of the design of any software system, and specialized security teams periodically review code for any potential software vulnerabilities. As this newer, more secure code reaches the market, and as more countermeasures are being deployed, attackers will find it increasingly difficult to attack a device through software vulnerabilities.

Software implementations have always been the weakest link in the security chain, but as the software design processes evolve, this is gradually becoming less the case. Attackers will have to find other means of circumventing the security measures that are in place. One way to do that, is to deceive the users of devices into willingly installing malicious software. Users are often not capable of making decisions that pertain to security, and as such it is up to the security research community to find ways that will keep users safe.

Frameworks such as the Security-by-Contract framework will become more important in the future. SxC does not rely on user input, and ensures that any application that is being deployed on the device will not violate a fixed system policy. Malicious applications might still be able to exploit some of the system resources, but only up to a measurable and predetermined point.

Future research will focus on 'empowering the user'. It will help the user to understand the security consequences of installing an application, and will guide him in making the right decisions. An attacker should not have *any* control over this process, as this might allow him to influence the user in some way.

Chip manufacturers are also improving support for hardware-based virtualization or application isolation. One particular example is the TrustZone API [8] for the ARM architecture. This API offers the primitives to build a secure subsystem that manages sensitive data such as cryptographic keys. Future research will leverage these new hardware components, and build trusted subsystems that further decrease the likelihood that sensitive data is stolen from the device, even in the event of a software compromise.

Appendix A

Alphanumeric ARM Instructions

The table below lists all the instructions present in ARMv6. For each instruction, we have checked some simple constraints that may not be broken in order for the instruction to be alphanumeric. The main focus of this table is the high order bits of the second byte of the instruction (bits 23 to 20). Only the high order bits of this byte are included, because the high order bits of the first byte are set by the condition flags, and the high order bits of the third and fourth byte are often set by the operands of the instruction. When the table contains the value 'd' for a bit, it means that the value of this bit depends on specific settings.

The final column contains a list of things that disqualify the instruction for being used in alphanumeric shellcode. Disqualification criteria are that at least one of the four bytes of the instruction is either always too high to be alphanumeric, or too low. In this column, the following conventions are used:

- 'IO' is used to indicate that one or more bits is always 1
- 'IZ' is used to indicate that one or more bits is always 0
- 'SO' is used to indicate that one or more bits should be 1
- 'SZ' is used to indicate that one or more bits should be 0

Table A.1: The ARMv6 instruction set

Instruction	Version	23	22	21	20	Disqualifiers
ADC		1	0	1	d	IO: 23
ADD		1	0	0	d	IO: 23
AND		0	0	0	d	IZ: 23-21
B, BL		d	d	d	d	
BIC		1	1	0	d	IO: 23
BKPT	5+	0	0	1	0	IO: 31, IZ: 22, 20
BLX (1)	5+	d	d	d	d	IO: 31
BLX (2)	5+	0	0	1	0	SO: 15, IZ: 22, 20
BX	4T, 5+	0	0	1	0	IO: 7, SO: 15, IZ 22, 20
BXJ	5TEJ, 6+	0	0	1	0	SO: 15, IZ: 22, 20, 6, 4
CDP		d	d	d	d	
CLZ	5+	0	1	1	0	IZ: 7-5
CMN		0	1	1	1	SZ: 15-13
CMP		0	1	0	1	SZ: 15-13
CPS	6+	0	0	0	0	SZ: 15-13, IZ 22-20
CPY	6+	1	0	1	0	IZ: 22, 20, 7-5, IO 23
EOR		0	0	1	d	
LDC		d	d	d	1	
LDM (1)		d	0	d	1	
LDM (2)		d	1	0	1	
LDM (3)		d	1	d	1	IO: 15
LDR		d	0	d	1	
LDRB		d	1	d	1	
LDRBT		0	1	1	1	
LDRD	5TE+	d	d	d	0	
LDREX	6+	1	0	0	1	IO: 23, 7
LDRH		d	d	d	1	IO: 7
LDRSB	4+	d	d	d	1	IO: 7
LDRSH	4+	d	d	d	1	IO: 7
LDRT		d	0	1	1	
MCR		d	d	d	0	
MCRR	5TE+	0	1	0	0	
MLA		0	0	1	d	IO: 7
MOV		1	0	1	d	IO: 23
MRC		d	d	d	1	
MRRC	5TE+	0	1	0	1	
MRS		0	d	0	0	SZ: 7-0
MSR		0	d	1	0	SO: 15
MUL		0	0	0	d	IO: 7
MVN		1	1	1	d	IO: 23

Continued on next page

Table A.1 – continued from previous page

Instruction	Version	23	22	21	20	Disqualifiers
ORR		1	0	0	d	IO: 23
PKHBT	6+	1	0	0	0	IO: 23
PKHTB	6+	1	0	0	0	IO: 23
PLD	5TE+, !5TExP	d	1	0	1	IO: 15
QADD	5TE+	0	0	0	0	IZ: 22-21
QADD16	6+	0	0	1	0	IZ: 22, 20
QADD8	6+	0	0	1	0	IZ: 22, 20, IO: 7
QADDSUBX	6+	0	0	1	0	IZ: 22, 20
QDADD	5TE+	0	1	0	0	
QDSUB	5TE+	0	1	1	0	
QSUB	5TE+	0	0	1	0	IZ: 22, 20
QSUB16	6+	0	0	1	0	IZ: 22, 20
QSUB8	6+	0	0	1	0	IZ: 22, 20, IO: 7
QSUBADDX	6+	0	0	1	0	IZ: 22, 20
REV	6+	1	0	1	1	IO: 23
REV16	6+	1	0	1	1	IO: 23, 7
REVSH	6+	1	1	1	1	IO: 23, 7
RFE	6+	d	0	d	1	SZ: 14-13, 6-5
RSB		0	1	1	d	
RSC		1	1	1	d	IO: 23
SADD16	6+	0	0	0	1	IZ: 22-21
SADD8	6+	0	0	0	1	IZ: 22-21, IO: 7
SADDSUBX	6+	0	0	0	1	IZ: 22-21
SBC		1	1	0	d	IO: 23
SEL	6+	1	0	0	0	IO: 23
SETEND	6+	0	0	0	0	SZ: 14-13, IZ: 22-21, 6-5
SHADD16	6+	0	0	1	1	IZ: 6-5
SHADD8	6+	0	0	1	1	IO: 7
SHADDSUBX	6+	0	0	1	1	
SHSUB16	6+	0	0	1	1	
SHSUB8	6+	0	0	1	1	IO: 7
SHSUBADDX	6+	0	0	1	1	
SMLA	5TE+	0	0	0	0	IO: 7, IZ: 22-21
SMLAD	6+	0	0	0	0	IZ: 22-21
SMLAL		1	1	1	d	IO: 23,7
SMLAL	5TE+	0	1	0	0	IO: 7
SMLALD	6+	0	1	0	0	
SMLAW	5TE+	0	0	1	0	IZ: 22, 20, IO: 7
SMLSD	6+	0	0	0	0	IZ: 22-21

Continued on next page

Table A.1 – continued from previous page

Instruction	Version	23	22	21	20	Disqualifiers
SMLSLD	6+	0	1	0	0	
SMMLA	6+	0	1	0	1	
SMMLS	6+	0	1	0	1	IO: 7
SMMUL	6+	0	1	0	1	IO: 15
SMUAD	6+	0	0	0	0	IZ: 22-21, IO: 15
SMUL	5TE+	0	1	1	0	SZ: 15, IO: 7
SMULL		1	1	0	d	IO: 23
SMULW	5TE+	0	0	1	0	IZ: 22, 20,SZ: 14-13, IO: 7
SMUSD	6+	0	0	0	0	IZ: 22-21, IO: 15
SRS	6+	d	1	d	0	SZ: 14-13, 6-5
SSAT	6+	1	0	1	d	IO: 23
SSAT16	6+	1	0	1	0	IO: 23
SSUB16	6+	0	0	0	1	IZ: 22-21
SSUB8	6+	0	0	0	1	IZ: 22-21, IO: 7
SSUBADDX	6+	0	0	0	1	IZ: 22-21
STC	2+	d	d	d	0	
STM (1)		d	0	d	0	IZ: 22, 20
STM (2)		d	1	0	0	
STR		d	0	d	0	IZ: 22, 20
STRB		d	1	d	0	
STRBT		d	1	1	0	
STRD	5TE+	d	d	d	0	IO: 7
STREX	6+	1	0	0	0	IO: 7
STRH	4+	d	d	d	0	IO: 7
STRT		d	0	1	0	IZ: 22, 20
SUB		0	1	0	d	
SWI		d	d	d	d	
SWP	2a, 3+	0	0	0	0	IZ: 22-21, IO: 7
SWPB	2a, 3+	0	1	0	0	IO: 7
SXTAB	6+	1	0	1	0	IO: 23
SXTAB16	6+	1	0	0	0	IO: 23
SXTAH	6+	1	0	1	1	IO: 23
SXTB	6+	1	0	1	0	IO: 23
SXTB16	6+	1	0	0	0	IO: 23
SXTH	6+	1	0	1	1	IO: 23
TEQ		0	0	1	1	SZ: 14-13
TST		0	0	0	1	IZ: 22-21, SZ: 14-13
UADD16	6+	0	1	0	1	IZ: 6-5
UADD8	6+	0	1	0	1	IO: 7
UADDSUBX	6+	0	1	0	1	

Continued on next page

Table A.1 – continued from previous page

Instruction	Version	23	22	21	20	Disqualifiers
UHADD16	6+	0	1	1	1	IZ: 6-5
UHADD8	6+	0	1	1	1	IO: 7
UHADDSUBX	6+	0	1	1	1	
UHSUB16	6+	0	1	1	1	
UHSUB8	6+	0	1	1	1	IO: 7
UHSUBADDX	6+	0	1	1	1	
UMAAL	6+	0	1	0	0	IO: 7
UMLAL		1	0	1	d	IO: 23, 7
UMULL		1	0	0	d	IO: 23, 7
UQADD16	6+	0	1	1	0	IZ: 6-5
UQADD8	6+	0	1	1	0	IO: 7
UQADDSUBX	6+	0	1	1	0	
UQSUB16	6+	0	1	1	0	
UQSUB8	6+	0	1	1	0	IO: 7
UQSUBADDX	6+	0	1	1	0	
USAD8	6+	1	0	0	0	IO: 23, 15, IZ: 6-5
USADA8	6+	1	0	0	0	IO: 23, IZ: 6-5
USAT	6+	1	1	1	d	IO: 23
USAT16	6+	1	1	1	0	IO: 23
USUB16	6+	0	1	0	1	
USUB8	6+	0	1	0	1	IO: 7
USUBADDX	6+	0	1	0	1	
UXTAB	6+	1	1	1	0	IO: 23
UXTAB16	6+	1	1	0	0	IO: 23
UXTAH	6+	1	1	1	1	IO: 23
UXTB	6+	1	1	1	0	IO: 23
UXTB16	6+	1	1	0	0	IO: 23
UXTH	6+	1	1	1	1	IO: 23

Bibliography

[1] Martin Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353, Alexandria, Virginia, U.S.A., November 2005. ACM.

[2] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, QC, August 2009.

[3] Irem Aktug and Katsiaryna Naliuka. ConSpec – a formal language for policy specification. In *Proceedings of the First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM2007)*, September 2007 (accepted).

[4] Aleph1. Smashing the stack for fun and profit. *Phrack*, 49, 1996.

[5] The Open Handset Alliance. Android security and permissions. http://developer.android.com/guide/topics/security/security.html.

[6] Alexander Anisimov. Defeating Microsoft Windows XP SP2 heap protection and DEP bypass.

[7] Anonymous. Once upon a free(). *Phrack*, 57, 2001.

[8] ARM. The trustzone security foundation. `http://www.arm.com/products/processors/technologies/trustzone.php`.

[9] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 290–301, Orlando, Florida, U.S.A., June 1994. ACM.

[10] Elena G. Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanović, and Dino D. Zovi. Randomized instruction set emulation

to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS2003)*, pages 281–289, Washington, D.C., U.S.A., October 2003. ACM.

[11] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with Polymer. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 305–314, June 2005.

[12] M. Becher and R. Hund. Kernel-level interception and applications on Windows Mobile devices. Reihe Informatik Tech. Rep. TR-2008-003.

[13] Sandeep Bhatkar, Daniel C. Duvarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, Washington, D.C., U.S.A., August 2003. USENIX Association.

[14] Sandeep Bhatkar and R. Sekar. Data space randomization. In *Proceedings of the 5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, volume 5137 of *Lecture Notes in Computer Science*, Paris, France, July 2008. Springer.

[15] Sandeep Bhatkar, R. Sekar, and Daniel C. Duvarney. Efficient techniques for comprehensive protection from memory error exploits. In *14th USENIX Security Symposium*, Baltimore, MD, August 2005. USENIX Association.

[16] N. Bielova, N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Matching in security-by-contract for mobile code. *Journal of Logic and Algebraic Programming*, 78(5):340–358, 2009.

[17] Blexim. Basic integer overflows. *Phrack*, 60, December 2002.

[18] Cédric Boon, Pieter Philippaerts, and Frank Piessens. Practical experience with the .NET cryptographic API. In *Benelux Workshop on Information and System Security*, 2008.

[19] A. Bose, X. Hu, K.G. Shin, and T. Park. Behavioral detection of malware on mobile handsets. In *Proceeding of the 6th international conference on Mobile systems, applications, and services*, pages 225–238. ACM, 2008.

[20] Eric Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *15th ACM Conference on Computer and Communications Security*, October 2008.

[21] Bulba and Kil3r. Bypassing Stackguard and stackshield. *Phrack*, 56, 2000.

[22] J. Cheng, S.H.Y. Wong, H. Yang, and S. Lu. SmartSiren: virus detection and alert for smartphones. In *Proceedings of the 5th international conference on Mobile systems, applications and services*, page 271. ACM, 2007.

[23] T. Chiueh and Fu H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 409–420, Phoenix, Arizona, USA, April 2001. IEEE Computer Society, IEEE Press.

[24] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[25] CNET. Armed for the living room. `http://news.cnet.com/ARMed-for-the-living-room/2100-1006_3-6056729.html`.

[26] Jeremy Condit, Matthew Harren, Scott Mcpeak, George C. Necula, and Westley Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 232–244, San Diego, California, U.S.A., 2003. ACM.

[27] Crispin Cowan, Steve Beattie, Ryan F. Day, Calton Pu, Perry Wagle, and Eric Walthinsen. Protecting systems from stack smashing attacks with StackGuard. In *Proceedings of Linux Expo 1999*, Raleigh, North Carolina, U.S.A., May 1999.

[28] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, Washington, D.C., U.S.A., August 2003. USENIX Association.

[29] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, San Antonio, Texas, U.S.A., January 1998. USENIX Association.

[30] DalvikVM.com. The Dalvik virtual machine. http://www.dalvikvm.com/.

[31] Lieven Desmet, Wouter Joosen, Fabio Massacci, Katsiaryna Naliuka, Pieter Philippaerts, Frank Piessens, and Dries Vanoverberghe. A flexible security architecture to support third-party applications on mobile devices. In *CSAW '07: Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 19–28, New York, NY, USA, 2007. ACM.

[32] Lieven Desmet, Fabio Massacci, and Katsiaryna Naliuka. Multisession monitor for .net mobile applications: theory and implementation. In *Nordic Workshop on Secure IT Systems (Nordsec 2007)*, October 2007.

[33] Igor Dobrovitski. Exploit for CVS double free() for linux pserver. http://seclists.org/lists/bugtraq/2003/Feb/0042.html, February 2003.

[34] Nicola Dragoni, Fabio Massacci, Katsiaryna Naliuka, Roberto Sebastiani, Ida Siahaan, Thomas Quillinan, Ilaria Matteucci, and Christian Schaefer. S3MS deliverable D2.1.4 - methodologies and tools for contract matching, April 2007.

[35] Nicola Dragoni, Fabio Massacci, Katsiaryna Naliuka, and Ida Siahaan. Security-by-contract: Toward a semantics for digital signatures on mobile code. In *EuroPKI*, pages 297–312, 2007.

[36] Riley Eller. Bypassing msb data filters for buffer overflow exploits on intel platforms, August 2000.

[37] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and Communications Security*, pages 235–245. ACM, 2009.

[38] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *IEEE Security & Privacy Magazine*, 7:10–17, 2009.

[39] Ulfar Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Cornell University, 2004. Adviser-Fred B. Schneider.

[40] Úlfar Erlingsson. Low-level software security: Attacks and defenses. Technical Report MSR-TR-2007-153, Microsoft Research, November 2007.

[41] Ulfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 246, Washington, DC, USA, 2000. IEEE Computer Society.

[42] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks. Technical report, IBM Research Divison, Tokyo Research Laboratory, June 2000.

[43] F-Secure. Just because it's signed doesn't mean it isn't spying on you. http://www.f-secure.com/weblog/archives/00001190.html.

[44] Cédric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. *ACM Trans. Program. Lang. Syst.*, 25(3):360–399, 2003.

[45] funkysh. Into my ARMs: Developing StrongARM/Linux shellcode. *Phrack*, 58, December 2001.

[46] Francesco Gadaleta, Yves Younan, and Wouter Joosen. BuBBle: A Javascript engine level countermeasure against heap-spraying attacks. In *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSOS)*, pages 1–17, Pisa, Italy, February 2010.

[47] Google. Android. `http://www.android.com/`.

[48] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Certified in-lined reference monitoring on .net. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 7–16, New York, NY, USA, 2006. ACM Press.

[49] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, July 2000.

[50] Tim Hurman. Exploring Windows CE shellcode, June 2005.

[51] ARM Inc. Arm architecture reference manua. http://www.arm.com/miscPDFs/14128.pdf, 2005.

[52] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, California, U.S.A., June 2002. USENIX Association.

[53] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging*, number 009-02 in Linköping Electronic Articles in Computer and Information Science, pages 13–26, Linköping, Sweden, 1997. Linköping University Electronic Press.

[54] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS2003)*, pages 272–280, Washington, D.C., U.S.A., October 2003. ACM.

[55] Samuel C. Kendall. Bcc: Runtime checking for C programs. In *Proceedings of the USENIX Summer 1983 Conference*, pages 5–16, Toronto, Ontario, Canada, July 1983. USENIX Association.

[56] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, California, U.S.A., August 2002. USENIX Association.

[57] Sven Köhler, Christian Schindelhauer, and Martin Ziegler. On approximating real-world halting problems. In *15th International Symposium on Fundamentals of Computation Theory*, volume 3623 of *Lecture Notes in Computer Science*, September 2005.

[58] Sumant Kowshik, Dinakar Dhurjati, and Vikram Adve. Ensuring code safety without runtime checks for real-time control systems. In *Proceedings of the International Conference on Compilers Architecture and Synthesis for Embedded Systems*, pages 288–297, Grenoble, France, October 2002.

[59] Andreas Krennmair. ContraPolice: a libc extension for protecting applications from heap-smashing attacks, November 2003.

[60] James R. Larus, Thomas Ball, Manuvir Das, Robert Deline, Manuel Fähndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, May /June 2004.

[61] Kyung S. Lhee and Steve J. Chapin. Type-assisted dynamic buffer overflow detection. In *Proceedings of the 11th USENIX Security Symposium*, pages 81–90, San Francisco, California, U.S.A., August 2002. USENIX Association.

[62] Kyung S. Lhee and Steve J. Chapin. Buffer overflow and format string overflow vulnerabilities. *Software: Practice and Experience*, 33(5):423–460, April 2003.

[63] ARM Limited. *ARM7TDMI Technical Reference Manual*. ARM Ltd., 2004.

[64] Symbian Ltd. Symbian signed. https://www.symbiansigned.com.

[65] J. Mason, S. Small, F. Monrose, and G. MacManus. English shellcode. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 524–533. ACM, 2009.

[66] F. Massacci and I. Siahaan. Matching midlet's security claims with a platform security policy using automata modulo theory. NordSec, 2007.

[67] Fabio Massacci and Katsiaryna Naliuka. Towards practical security monitors of uml policies for mobile applications. *International Conference on Availability, Reliability and Security*, pages 1112–1119, 2008.

[68] Stephen Mccamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, British Columbia, Canada, August 2006. USENIX Association.

[69] Microsoft. The .NET framework developer center. http://msdn.microsoft.com/en-us/netframework/.

[70] Microsoft. The shared source CLI (rotor). http://www.microsoft.com/resources/sharedsource/.

[71] Microsoft. Understanding the windows mobile security model. http://technet.microsoft.com/en-us/library/cc512651.aspx.

[72] H. D. Moore. Cracking the iPhone. http://blog.metasploit.com/2007/10/cracking-iphone-part-1.html.

[73] Urban Müller. Brainf*ck. http://www.muppetlabs.com/ breadbox/bf/, June 1993.

[74] D. Muthukumaran, A. Sawani, J. Schiffman, B.M. Jung, and T. Jaeger. Measuring integrity on mobile phone systems. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 155–164. ACM, 2008.

[75] National Institute of Standards and Technology. National vulnerability database statistics. http://nvd.nist.gov/statistics.cfm.

[76] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Prgramming Language Design and Implementation (PLDI)*, pages 333–344, 1998.

[77] George Necula, Scott Mcpeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, Portland, Oregon, U.S.A., January 2002. ACM.

[78] Novell. Mono: Cross platform, open source .net development framework. http://www.mono-project.com/.

[79] Yutaka Oiwa, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. Fail-safe ANSI-C compiler: An approach to making C programs secure: Progress report. In *Proceedings of International Symposium on Software Security 2002*, pages 133–153, Tokyo, Japan, November 2002.

[80] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *2009 Annual Computer Security Applications Conference*, pages 340–349. IEEE, 2009.

[81] Tavis Ormandy. LibTIFF next RLE decoder remote heap buffer overflow vulnerability. http://www.securityfocus.com/bid/19282, Aug 2006.

[82] Tavis Ormandy. LibTIFF TiffFetchShortPair remote buffer overflow vulnerability. http://www.securityfocus.com/bid/19283, Aug 2006.

[83] Alfredo Ortega. Android web browser gif file heap-based buffer overflow vulnerability, March 2008.

[84] Harish Patil and Charles N. Fischer. Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs. *Software: Practice and Experience*, 27(1):87–110, January 1997.

[85] Pieter Philippaerts, Cédric Boon, and Frank Piessens. Report: Extensibility and implementation independence of the .NET cryptographic API. In *Lecture Notes in Computer Science*, volume 5429, pages 101–110. Springer, 2009.

[86] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–272, Washington, D.C., U.S.A., August 2003. USENIX Association.

[87] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. Nozzle: A defense against heap-spraying code injection attacks. Technical Report MSR-TR-2008-176, Microsoft Research, November 2008.

[88] Bill Ray. Symbian signing is no protection from spyware. `http://www.theregister.co.uk/2007/05/23/symbian_signed_spyware`, May 2007.

[89] A. Refsdal, B. Solhaug, and K. Stølen. A UML-based method for the development of policies to support trust management. *Trust Management II*, pages 33–49, 2008.

[90] Gerardo Richarte. Four different tricks to bypass stackshield and stackguard protection, June 2002.

[91] Juan Rivas. Overwriting the .dtors section. Posted on the Bugtraq mailinglist http://www.securityfocus.com/archive/1/150396, 2000.

[92] rix. Writing IA32 alphanumeric shellcodes. *Phrack*, 57, August 2001.

[93] William Robertson, Christopher Kruegel, Darren Mutz, and Frederik Valeur. Run-time detection of heap-based overflows. In *Proceedings of the 17th Large Installation Systems Administrators Conference*, pages 51–60, San Diego, California, U.S.A., October 2003. USENIX Association.

[94] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, San Diego, California, U.S.A., February 2004. Internet Society.

[95] S3MS. Security of software and services for mobile systems. `http://www.s3ms.org/`, 2007.

[96] A.D. Schmidt, F. Peters, F. Lamour, C. Scheel, S.A. Çamtepe, and Ş. Albayrak. Monitoring smartphones for anomaly detection. *Mobile Networks and Applications*, 14(1):92–106, 2009.

[97] Scut. Exploiting format string vulnerabilities. http://www.team-teso.net/articles/formatstring/, 2001.

[98] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, Washington, D.C., U.S.A., October 2007. ACM, ACM Press.

[99] Hovav Shacham, Matthew Page, Ben Pfaff, Eu J. Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, Washington, D.C., U.S.A., October 2004. ACM, ACM Press.

[100] skape and Skywing. Bypassing Windows hardware-enforced data execution prevention. *Uninformed*, 2, September 2005.

[101] SkyLined. Internet Explorer IFRAME src&name parameter bof remote compromise, 2004.

[102] Andrew Sloss, Dominic Symes, and Chris Wright. *ARM System Developer's Guide*. Elsevier, 2004.

[103] Solar Designer. Getting around non-executable stack (and fix). Posted on the Bugtraq mailinglist http://www.securityfocus.com/archive/1/7480, August 1997.

[104] B. Solhaug, D. Elgesem, and K. Stolen. Specifying Policies Using UML Sequence Diagrams–An Evaluation Based on a Case Study. In *Proceedings of the Eighth IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 19–28. IEEE Computer Society, 2007.

[105] Alexander Sotirov. Reverse engineering and the ANI vulnerability, April 2007.

[106] Alexander Sotirov and Mark Dowd. Bypassing browser memory protections: Setting back browser security by 10 years. In *BlackHat 2008*, August 2008.

[107] Nora Sovarel, David Evans, and Nathanael Paul. Where's the FEEB? the effectiveness of instruction set randomization. In *Proceedings of the 14th USENIX Security Symposium*, Baltimore, Maryland, U.S.A., August 2005. Usenix.

[108] Joseph L. Steffen. Adding run-time checking to the portable C compiler. *Software: Practice and Experience*, 22(4):305–316, April 1992.

[109] Jon Stokes. ARM attacks Atom with 2GHz A9; can servers be far behind?

[110] Raoul Strackx, Yves Younan, Pieter Philippaerts, and Frank Piessens. Efficient and effective buffer overflow protection on arm processors. In *Information Security Theory and Practices: Security and Privacy of Pervasive Systems and Smart Devices*, pages 1–16. Springer, April 2010.

[111] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the European Workshop on System Security (Eurosec)*, Nuremberg, Germany, March 2009.

[112] Sun. The java platform. http://java.sun.com/.

[113] Sun. The mobile information device profile. http://java.sun.com/javame/index.jsp.

[114] The PaX Team. Documentation for the PaX project.

[115] D. Vanoverberghe and F. Piessens. Security enforcement aware software development. *Information and Software Technology*, 51(7):1172–1185, 2009.

[116] Dries Vanoverberghe and Frank Piessens. A caller-side inline reference monitor for an object-oriented intermediate language. In *Formal Methods for Open Object-based Distributed Systems (FMOODS)*, pages 240–258, 2008.

[117] Dries Vanoverberghe and Frank Piessens. Security enforcement aware software development. *Information and Software Technology*, 2009.

[118] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 203–216, Asheville, North Carolina, U.S.A., December 1993. ACM.

[119] Rafal Wojtczuk. Defeating solar designer non-executable stack patch. Posted on the Bugtraq mailinglist, February 1998.

[120] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. In *22nd International Symposium on Reliable Distributed Systems (SRDS'03)*, pages 260–269, Florence, Italy, October 2003. IEEE Computer Society, IEEE Press.

[121] Wei Xu, Daniel C. Duvarney, and R. Sekar. An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 117–126, Newport Beach, California, U.S.A., October-November 2004. ACM, ACM Press.

[122] Yves Younan. *Efficient Countermeasures for Software Vulnerabilities due to Memory Management Errors*. PhD thesis, Katholieke Universiteit Leuven, May 2008.

[123] Yves Younan, Wouter Joosen, and Frank Piessens. Code injection in C and C++ : A survey of vulnerabilities and countermeasures. Technical Report CW386, Departement Computerwetenschappen, Katholieke Universiteit Leuven, July 2004.

[124] Yves Younan and Pieter Philippaerts. Alphanumeric RISC ARM shellcode. *Phrack*, 66, June 2009.

[125] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. Paricheck: An efficient pointer arithmetic checker for C programs. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Bejing, China, April 2010. ACM.

[126] Yves Younan, Davide Pozza, Frank Piessens, and Wouter Joosen. Extended protection against stack smashing attacks without performance loss. In *Proceedings of the Twenty-Second Annual Computer Security Applications Conference (ACSAC '06)*, pages 429–438. IEEE Press, December 2006.

[127] X. Zhang, O. Acicmez, and J.P. Seifert. A trusted mobile phone reference architecturevia secure kernel. In *Conference on Computer and Communications Security: Proceedings of the 2007 ACM workshop on Scalable trusted computing*. Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA„ 2007.

Curriculum Vitae

Pieter obtained his masters degree in computer science at the K.U.Leuven in July 2004, and finished a second master in the subsequent year, specializing in Artificial Intelligence. Before joining the K.U.Leuven as a PhD student, Pieter worked as a .NET technical consultant.

Pieter is a member of the DistriNet research group, under the supervision of Frank Piessens and Wouter Joosen. The topic of his PhD thesis is mobile phone security. He has experience with both high- and low-level programming, and is particularly interested in application security and building new security architectures and mechanisms.

Relevant Publications

Conference Proceedings

- L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe. A flexible security architecture to support third-party applications on mobile devices. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, November 2007.
- L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe. The S3MS.NET run time monitor: Tool demonstration. In *Electronic Notes in Theoretical Computer Science*, March 2009.
- Y. Younan, P. Philippaerts, F. Piessens, W. Joosen, S. Lachmund, and T. Walter. Filter-resistant code injection on ARM. In *Proceedings of the 16th ACM conference on Computer and Communications Security*, November 2009.

Journals

- L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-contract on the .NET platform. In *Information security technical report*, May 2008.
- Y. Younan, P. Philippaerts, F. Piessens, W. Joosen, S. Lachmund, and T. Walter. Filter-resistant Code Injection on ARM. Accepted in *Journal in Computer Virology*, not yet published.

Book Chapters

- B. De Win, T. Goovaerts, W. Joosen, P. Philippaerts, F. Piessens, and Y. Younan. Security middleware for mobile applications. In *Middleware for network eccentric and mobile applications*, 2009.

- L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. A security architecture for Web 2.0 applications. In *Towards the Future Internet - A European Research Perspective*, 2009.

Patents

- P. Philippaerts, Y. Younan, F. Piessens, S. Lachmund, and T. Walter. Method and apparatus for preventing modification of a program execution flow. Patent Application no. 09161239.0-1245, Filing date 27/05/2009

Others

- Y. Younan, P. Philippaerts, and F. Piessens. Alphanumeric RISC ARM shellcode. In *Phrack Magazine*, June 2009.

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Computer Science
Research group DistriNet
Celestijnenlaan 200A, B-3001 Leuven